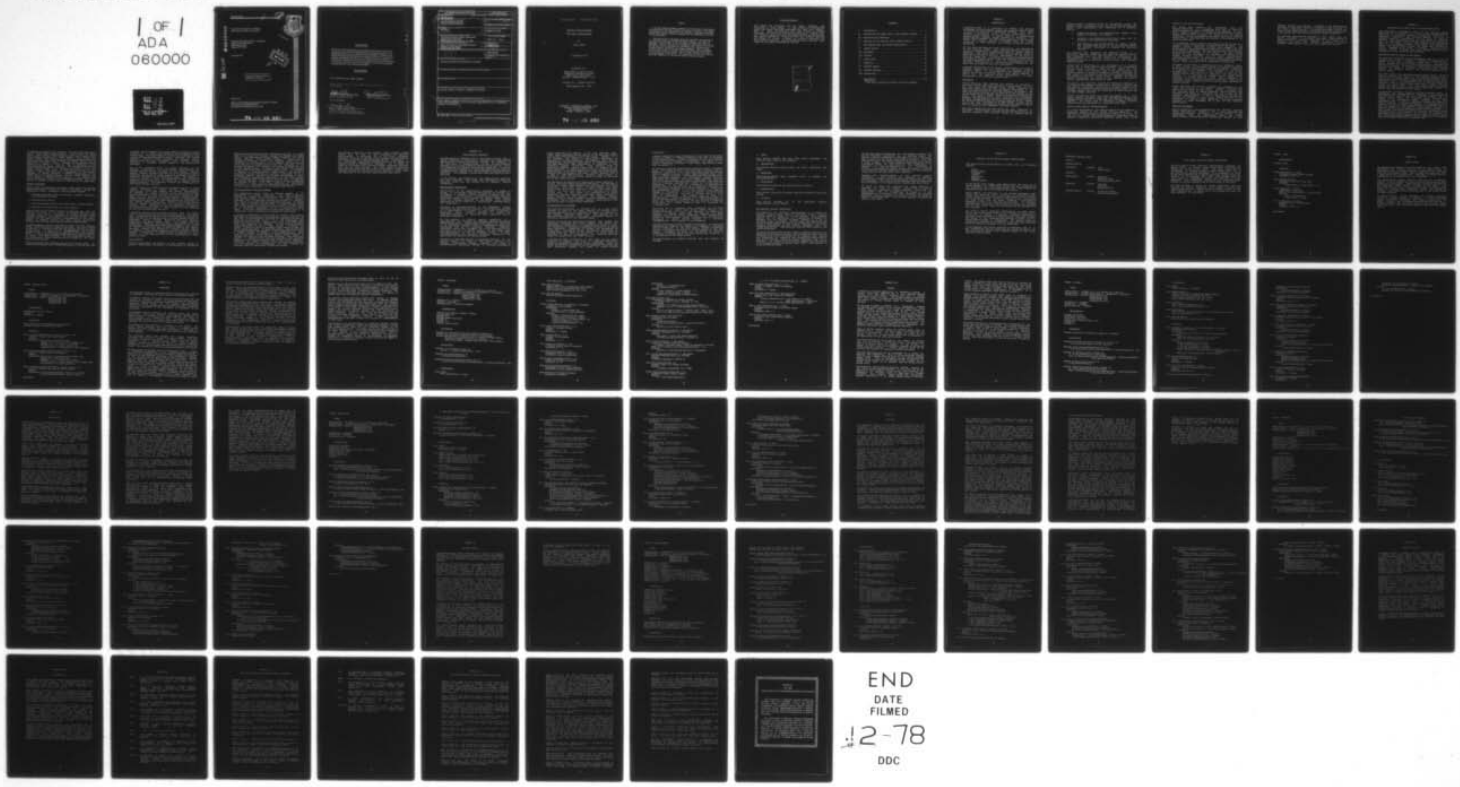
AD-A060 000 HONEYWELL INFORMATION SYSTEMS INC MCLEAN VA FEDERAL --ETC F/G 17/2
MULTICS SECURITY KERNEL TOP LEVEL SPECIFICATION.(U)
NOV 76 J STERN F19628-74-C-0193

UNCLASSIFIED ESD-TR-76-368 NL

1 OF 1
ADA
060000



END
DATE
FILMED
12-78
DDC

ESD-TR-76-368

MULTICS SECURITY KERNEL TOP LEVEL SPECIFICATION

Honeywell Information Systems, Incorporated
Federal Systems Operations
7900 Westpark Drive
McLean, VA 22101

November 1976

Approved for Public Release;
Distribution Unlimited.

Prepared for

DEPUTY FOR COMMAND AND MANAGEMENT SYSTEMS
ELECTRONIC SYSTEMS DIVISION
HANSCOM AIR FORCE BASE, MA 01731

LEGAL NOTICE

When U.S. Government drawings, specifications or other data are used for any purpose other than a definitely related government procurement operation, the government thereby incurs no responsibility nor any obligation whatsoever; and the fact that the government may have formulated, furnished, or in any way supplied the said drawings, specifications, or other data is not to be regarded by implication or otherwise as in any manner licensing the holder or any other person or conveying any rights or permission to manufacture, use, or sell any patented invention that may in any way be related thereto.

OTHER NOTICES

Do not return this copy. Retain or destroy.

"This technical report has been reviewed and is approved for publication."

William R. Price

WILLIAM R. PRICE, Captain, USAF
Project Engineer/Scientist

Donald P. Eriksen

DONALD P. ERIKSEN
Project Engineer/Scientist

FOR THE COMMANDER

Frank J. Emma

FRANK J. EMMA, Colonel, USAF
Director, Computer Systems Engineering
Deputy for Command & Management Systems

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

(19) REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER (18) ESD-TR-76-368	2. GOVT ACCESSION NO.
3. RECIPIENT'S CATALOG NUMBER	
4. TITLE (and Subtitle) (6) MULTICS SECURITY KERNEL TOP LEVEL SPECIFICATION	5. TYPE OF REPORT & PERIOD COVERED (9) Technical Report
6. PERFORMING ORG. REPORT NUMBER	
7. AUTHOR(s) (10) Jerry Stern	8. CONTRACT OR GRANT NUMBER(s) (15) F19628-74-C-0193
9. PERFORMING ORGANIZATION NAME AND ADDRESS Honeywell Information Systems, Incorporated Federal Systems Operations 7900 Westpark Drive, McLean, VA 22101	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS CDRL Item A008
11. CONTROLLING OFFICE NAME AND ADDRESS (11) Deputy for Command and Management Systems Electronic Systems Division Hanscom AFB, MA 01731	12. REPORT DATE November 1976
	13. NUMBER OF PAGES 74
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) (12) 75p.	15. SECURITY CLASS. (of this report) UNCLASSIFIED
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A

16. DISTRIBUTION STATEMENT (of this Report)

Approved for Public Release; Distribution Unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Security kernel, Multics, computer security.

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This report presents results of initial engineering investigations into the development of a top-level specification of a security kernel for Multics.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

409 690

AB

P R O J E C T G U A R D I A N

MULTICS SECURITY KERNEL

TOP LEVEL SPECIFICATION

by

Jerry Stern

5 November 1976

prepared for

Department of the Air Force
Electronic Systems Division
Hanscom Air Force Base
Bedford, Massachusetts 01731

Contract No. F19628-74-C-0193

CDRL Sequence No. A008

Honeywell Information Systems, Inc.
Federal Systems Operations
7900 Westpark Drive
McLean, Virginia 22101

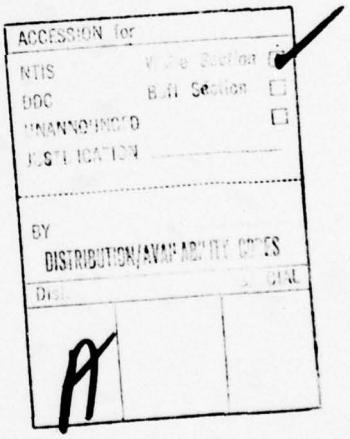
Preface

Air Force Systems Command terminated the effort which this document describes before the effort reached its logical conclusion. This report is incomplete but was published in the interest of capturing and disseminating the computer security technology that was available at the time of the termination.

This report was to precisely define the functional characteristics of a security kernel for the Multics computer system. The report is thorough in its treatment of the functional characteristics that are addressed. However, the report is incomplete. Many areas of the design are omitted, such as external I/O, reconfiguration, initialization, "trusted subject" interfaces, and message segments. Specific design decisions are avoided in certain areas; particularly various instances of the "finite resource problem". Most of these omissions are explicitly acknowledged in the report. The termination of this effort did not allow the completion of the security kernel functional design.

ACKNOWLEDGEMENTS

This report has benefited from the ideas, comments, and criticisms of many people. A long history of conversations between the author and Andre Bensoussan has been very helpful. Some of these conversations were also joined by Michael Schroeder. The author is particularly grateful to Douglas Hunt who reviewed early drafts of the specifications and contributed many useful suggestions. Thanks are also due Richard Feiertag who helped to resolve a number of problems regarding the specification technique. Bernard Greenberg was relied upon many times for his expert knowledge of Multics.



CONTENTS

I	Introduction	1
II	Reconciling the Model with a Real Computer System	5
III	Specification Technique	10
IV	Overview of the Multics Kernel Specification	15
V	The System Clock and Unique Identifiers	17
VI	Access Levels	19
VII	Processes	21
VIII	Volumes	27
IX	Quota Cells	33
X	Segments	41
XI	Address Spaces	51
XII	Message Segments	60
XIII	Conclusions	61
	References	62
	Appendix A Air Force Electronic Systems Division Comments	

CHAPTER I

INTRODUCTION

A security kernel is a collection of hardware and software mechanisms within a computer system that together control access to information according to prescribed policies. This report presents a preliminary formal top-level specification of a security kernel designed to support Multics, a sophisticated general-purpose computer system produced by Honeywell. The specifications describe the input/output behavior of the security kernel and, thereby, provide a suitable basis for formal validation of this behavior with respect to desired security properties.

The top-level specification plays a key role in the development of the security kernel. The top-level specification must be proven to correspond to a mathematical model of secure computer operation. The model chosen for the Multics security kernel is specifically devised to meet the requirements of the military security system [BL]. A technique for proving correspondence of this model to specifications of the kind presented in this report has been demonstrated [Mil]. Once the correspondence of the top-level specification to the model has been established, the model need no longer be explicitly considered.

A general methodology for the design, implementation and proof of software systems has been developed at Stanford Research Institute [RLNS]. It is believed that this methodology can be applied to the Multics security kernel, thus making possible a proof that the security kernel implementation, in fact, corresponds to the top-level specification.

Background

The Air Force has been studying the problem of providing a certifiably secure multilevel system for several years. In 1970, the Air Force Data Services Center (AFDSC) requested the Electronic Systems Division (ESD) to support development of an open multilevel system for the AFDSC Honeywell 635 systems. The resulting studies pointed out the severity of the problem and led to the formation of a computer security technology planning study panel. The panel's report [And] described the fundamental problems and delineated a program to develop the desired system. The panel recommended that the technical approach to the problem be "to start with a statement of an ideal system, a model, and to refine and move the statement through various levels of design into the mechanism that implements the model system".

The basic component of the ideal system was also identified by this panel. This component is known as the Reference Monitor, an abstract mechanism that controls access of subjects (active

system elements) to objects (units of information) within the computer system according to the rules of the military security system. Three requirements were recognized for a Reference Monitor:

a. Complete Mediation - the mechanism must mediate every access of a subject to an object.

b. Isolation - the mechanism and its data bases must be protected from unauthorized alteration.

c. Verifiability - the mechanism must be small, simple, and understandable so that it can be completely tested and verified (certified) to perform its functions correctly.

The mechanism that implements the Reference Monitor in a particular computer system has been termed the security kernel. Much subsequent work has been devoted to identifying the characteristics of a security kernel and to exploring the technology involved in producing a security kernel for a suitable computer system.

A major goal of the project is the development of a security kernel for a large resource sharing system. The system chosen for this effort is Multics. There are two reasons for this choice. First, the hardware base of the Multics system, the Honeywell Series 60 (Level 68) computer, has been identified as best suited of all commonly available large computer systems for the support of a security kernel [Smi]. Second, the Multics system architecture was conceived and developed with access control requirements specifically in mind.

One project, now completed, involved the design and production of a Multics system capable of supporting a two-level (Secret and Top Secret) environment for the Air Force Data Services Center [WBGHKS]. This system implements security controls based on the military access rules, but the correctness of these controls has not been formally verified.

Design of a security kernel for Multics was started as a joint effort between personnel from ESD, the MITRE Corporation, the Massachusetts Institute of Technology, and Honeywell Information Systems. The first step in producing a formal top-level specification was undertaken by a team from MITRE [SBB].

Nature of the Multics Security Kernel

To a first approximation, the security kernel can be viewed as a primitive nucleus of the current Multics supervisor. The supervisor is not replaced by the kernel; rather it is built on top of the kernel. This kernel-supervisor combination must be capable of supporting the Multics operating system with little

change to the user interface.

The kernel must contain all mechanisms that are security-sensitive, i.e., mechanisms that must be proven to perform properly in order to guarantee secure behavior. In this case, "secure behavior" is defined by the mathematical model. Certain access control policies now enforced by the Multics supervisor, but not addressed by the model, need not be enforced by the kernel. These policies can continue to be enforced by an unproven supervisor.

It is essential that the size of the security kernel be kept to a minimum in order to facilitate its specification and proof. The present Multics supervisor contains many examples of the interweaving of security-sensitive functions with other functions of no security importance. The security-sensitive functions must be extricated and isolated in the kernel. The most notable example of this philosophy is provided by the absence of the Multics directory hierarchy from the kernel specifications presented in this report. As proposed by Andre Bensoussan [Ben], the management of directories can be accomplished outside the kernel. Thus, the kernel has no knowledge of directories.

Isolation of the kernel is accomplished by two different means. A major portion of the kernel resides in a protected subsystem whose execution is distributed across all processes. Protection is provided by the hardware ring mechanism of the Multics processor [SS]. This portion of the kernel is akin to the current Multics ring 0 and ring 1 supervisor. For implementation reasons, it is convenient to isolate certain other kernel functions in single processes rather than attempting to distribute these functions across all processes. These kernel processes closely resemble some of the system daemon processes of current Multics.

Kernel processes, like ordinary processes, are supported by the inner ring portion of the kernel. However, the inner ring portion of the kernel provides certain special interfaces that are available only to kernel processes. These interfaces are internal to the kernel and therefore are not a part of the top-level specification. A few of these special interfaces, however, are included in this report simply to aid in the exposition of overall kernel operation. External interfaces provided by kernel processes are a part of the top-level specification.

Plan of this Report

Before proceeding to the presentation of the top-level security kernel specification, chapters 2 and 3 first introduce some prerequisite material. In chapter 2, certain aspects of the relationship among the mathematical model, the formal specifications, and the practical constraints imposed by real

computer systems are explored. In chapter 3, the specification technique is described. Chapter 4 provides a brief overview of the kernel specifications. The remaining chapters contain the actual formal specifications and accompanying prose descriptions. A familiarity with the Multics system is assumed throughout.

The top-level specification presented in this report does not specify interfaces to the kernel I/O system. Specification of these interfaces is awaiting evaluation of the results of a separate I/O study [Hon]. Operator interfaces to perform hardware reconfiguration have also been temporarily omitted.

CHAPTER II

RECONCILING THE MODEL WITH A REAL COMPUTER SYSTEM

Two fundamental requirements exist for the security kernel described in this report: (1) it must satisfy the mathematical model, and (2) it must be capable of supporting Multics in a reasonably compatible and efficient manner. As might be expected, these two requirements are not always in harmony. Moreover, considerations that have no relevance in the simplistic world of the model can be of great concern in the complicated world of a real computer system. This chapter first attempts to sketch the relationship between the model and the security kernel. Knowledge of the model is assumed. Following this, certain general problems that real systems must face in attempting to meet the model are discussed.

Relationship of the Model to the Kernel

The principal components of the mathematical model are a set of subjects and a set of objects. A subject is a surrogate for a person, in this case, a computer system user. The security kernel employs the Multics process abstraction to represent the notion of a subject. An object is an information container which may take any number of different forms within the computer system. For example, segments are objects supported by the Multics kernel.

The model embodies two non-discretionary access control policies known as security and integrity. The integrity policy was not a feature of the original model, but has since been adopted [Bib]. Each subject and each object possess both a security level and an integrity level. For simplicitly, both of these attributes are combined in the top-level specification into a single attribute called an access level. The relationship between a subject access level and an object access level determines whether the subject can read and/or write the object.

An additional attribute, called the visibility access level, is associated with certain objects. Detection of the existence of such objects is controlled by this attribute rather than the regular access level. The visibility access level is not a feature of the mathematical model. However, it has been introduced here in order to explicitly address the problem of object detection. In certain situations, it is appropriate for a process to be able to detect the existence of an object and yet have no access to the contents of the object. Therefore, a distinct access level is needed to control object detection. In most cases, the visibility access level of an object will equal the access level of its creator. In addition to detection of existence, the visibility access level is also the logical choice for controlling object deletion and the observation of permanent object attributes specified at creation time.

In addition to security and integrity, the model also describes a form of discretionary access control. There exists an access matrix, each entry of which describes the mode of access that a given subject may have for a given object. This access mode cannot exceed that dictated by the non-discretionary policies. Aside from this restriction, however, the access matrix may be arbitrarily modified. Thus, a process can always grant itself access to an object so long as this is permitted by the non-discretionary access controls. The standard discretionary access control mechanism now provided by Multics, i.e., the access control list (ACL), is more restrictive than the model requires. Hence, ACLs need not be implemented by the kernel. Instead, the kernel maintains a simpler representation of the access matrix for segment objects. For all other objects, the access matrix is degenerate and static, i.e., all subjects have full discretionary access to all non-segment objects. Any finer control of access to these objects is provided by the supervisor.

Trusted Processes

There exists a collection of functions needed within the computer system environment that have no direct counterpart within the mathematical model. Examples of such functions include:

1. changing the access levels of serially reusable resources, e.g. peripheral devices;

2. accepting user logins;

3. repairing damaged kernel data bases after a system failure;

4. changing the access level of a user segment.

These operations are clearly critical to security, yet they are outside the scope of the model. To preserve security, these functions must only be performed by trusted persons using verified programs. Note, however, that the verification of such programs does not entail proving a correspondence to the model. Rather, it requires proving that each program performs its specific task correctly and without unwanted side-effects.

It seems helpful to draw a distinction between the basic security kernel, which enforces the rules of the model, and this second collection of "trusted" functions that operate outside the realm of the model. In terms of an implementation, the trusted functions are, to a large extent, supported by the basic security kernel. In fact, many ordinary kernel functions can also operate on behalf of trusted functions. The only difference is that in the trusted mode, kernel functions are not subject to the usual security restrictions.

Trusted functions must interface directly to trusted users. No uncertified programs can be interposed between a trusted user and

a trusted function because this could result in misuse of trusted functions in a manner not intended by and not apparent to the trusted user. This requirement for direct user interaction shows a clear contrast between security kernel functions and trusted functions. Security kernel functions are primitive and far removed from the user interaction level whereas trusted functions, by necessity, cannot depend on any higher level functions.

Due to the different nature of trusted functions and the different requirements for verification, specifications for trusted functions are not provided in this report. However, because trusted functions can make use of more primitive kernel functions, the ability of kernel functions to operate in either a trusted or untrusted mode is explicitly shown. The method by which the kernel is able to discriminate between trusted and untrusted use of functions is by means of a special trusted process attribute. This attribute is taken into consideration in each access control decision.

A simple scheme for handling trusted processes would be to allow trusted processes to bypass entirely normal security restrictions. This scheme, however, has certain disadvantages. The examples of trusted functions given previously included resource management, error recovery, and other operations that might best be delegated to a small number of trusted persons. Security matters would be the principal concern of these persons, called security officers, who must necessarily be trustworthy to the maximum system access level. However, there exists another class of trusted operations that are regularly required by ordinary users. Insisting that all such operations be performed by a security officer places a heavy burden on the security officer and inconveniences the user.

A diffenent approach to this problem is taken here. An access level is still associated with each trusted process. This access level determines the degree to which the process can be trusted. Thus, any user can be provided with a trusted process capable of performing certain trusted functions, but still properly restricted by the user's access level. This agrees with the fundamental assumption that each user is trustworthy to his own access level. When executing unverified programs, a user cannot be trusted and therefore must be forced to abide by all rules of the model. However, when confined to a trusted process capable of executing only trusted programs, certain restrictions can be removed. In particular, a trusted process is permitted to write objects of a lower security level and to read objects of a lower integrity level.

Time Channels

With the given model, the ability of the security kernel to prevent unauthorized disclosure or modification of information

depends upon the assumption that information can only be passed from one subject to another through some commonly accessible object. Such information channels between subjects are called storage channels. There exists, however, a second class of information channels, called time channels, which are not accounted for by the model. If, for example, the time required for a subject S1 to access object O1 can be influenced by the decision of subject S2 to access or not access an object O2, then a time channel exists. In terms of real systems, time channels almost always exist when a physical resource (e.g., a processor or memory) is time-multiplexed among users. The control or elimination of time channels is generally recognized to be a difficult problem, the solution to which would appear to impose an unacceptable performance degradation on any real system. There is perhaps some consolation in the fact that, in comparison to storage channels, time channels tend to be low in bandwidth and difficult to use. In any case, prevention of unauthorized disclosure or modification of information through time channels is not a formal requirement for the kernel-based Multics.

The Shared Finite Resource Problem

The mathematical model, being an abstract system, is not constrained by physical limitations. The number of abstract objects, for example, can be arbitrarily large. The situation in real computer systems is quite different, of course. Physical resources such as secondary storage, main memory, I/O channels, etc. exist in limited sizes and quantities that must be shared among users of different access levels. A general problem arises when a user requests a particular resource that is in use, filled up, or otherwise unavailable due to the activities of other users. If the user requesting the resource can learn that it is unavailable, then an uncontrolled information path exists.

A variety of solutions to this problem are known. Two general approaches are preallocation and automatic time-multiplexing. In the case of preallocation, resources are statically partitioned among the various access levels. Thus, the information path described above can only be exploited by users of the same access level and therefore is harmless. In the case of automatic time-multiplexing, the user is never refused a resource, but rather is forced to wait (a presumably short length of time) for one to become available. As mentioned earlier, time-multiplexing inevitably produces time channels. A third approach is to allow a trusted process to manage a given shared resource. Note, however, that such a trusted process cannot simply respond to requests for resources from untrusted processes. This would only succeed in reproducing the very same information path described above with the trusted process serving as part of the mechanism. Therefore, the trusted process must be externally controlled, e.g. by an operator or administrator, and user requests for resources must be communicated outside the system.

The problem of sharing a finite resource among multiple access levels will be seen to recur throughout the top-level specification. In principle, any of the three techniques described above can be applied to any instances of this problem. In practice, however, these approaches may have undesirable, if not intolerable, operational characteristics, or may require an excessive amount of mechanism to implement. For a number of cases of the shared finite resource problem, acceptable solutions have not yet been devised. For these cases, the top-level specification circumvents the problem by essentially treating the resource involved as infinite in size or quantity. These cases are identified in the top-level specification description.

CHAPTER III

SPECIFICATION TECHNIQUE

The specification technique used in this report is based upon a method suggested by Parnas [Par]. Although Parnas specifically addresses the "software engineering" problem, the technique is not restricted to software alone. In fact, the specifications presented in this report describe aspects of the security kernel that encompass both hardware and software mechanisms. A notation is employed that conforms to a formal specification and assertion language known as SPECIAL that was developed at Stanford Research Institute. The language is relatively young and therefore still evolving.

In this chapter, the objectives of the specification technique are first examined. Next some basic features of SPECIAL are described along with the concept of an abstract machine interpreter.

Specification Objectives

The objective of the specification technique, as stated by Parnas, is to provide a precise specification of a program (or, in general, an abstract machine) without revealing too much information. In particular, a specification should supply the information needed to use a program, but should reveal nothing about the internal operation of the program. Thus, a Parnas specification can be viewed as an input/output characterization of a "black box".

Avoidance of over-specification is the fundamental concept underlying the specification technique. The information provided by a specification is limited to that which is externally observable. Parnas, however, expresses no concern for under-specification, i.e. providing less information than is externally observable.

For the purpose of proving security properties of a specification, the concerns are somewhat different from those emphasized by Parnas. Under-specification cannot be tolerated. It is absolutely essential that a specification contain all information that can be externally observed. Otherwise, information paths not accounted for by the specifications may exist and will not be proven secure. On the other hand, over-specification is only of secondary concern. At worst, over-specification may introduce irrelevant information that complicates the proof of the specifications.

Avoidance of under-specification is extremely difficult for the current Multics. If viewed in full detail, Multics is an entirely deterministic system. However, this view is extraordinarily complicated because it includes knowledge of

hidden mechanisms (e.g. paging). At the user interface level, these hidden mechanisms can be ignored for the most part without sacrificing a deterministic view of the system. However, in a few instances, the effects of these hidden mechanisms are detectable at the user interface. For example, a zero page is automatically deallocated at the time it is selected for removal from main memory and this deallocation is reflected in user observable segment attributes. Such behavior cannot be completely specified without introducing the details of the paging mechanism. It seems possible to write a non-deterministic specification which states simply that a page may be deallocated sometime after it becomes a zero page. Clearly, however, this is a case of under-specification. Moreover, there exists an information path (of the storage channel variety) not accounted for by this specification.

Two approaches are apparent for dealing with those "features" of Multics that make hidden mechanisms detectable. One approach is to provide complete specifications that fully expose the hidden mechanisms. This would result in gross complexity that would hinder proofs of security properties. Even if this were done, however, it would only lead to the discovery of an already obvious fact: in most cases, the detectability of hidden mechanisms produces insecure information paths. Therefore, a second approach has been adopted. This approach requires changing Multics where necessary to prevent detection of hidden mechanisms. One can then provide complete specifications that show no evidence of these hidden mechanisms.

An Introduction to SPECIAL

The following description is intended only as a very brief overview of some of the basic features of SPECIAL. For a complete description of the language, the reader is referred to the SPECIAL reference manual [RR]. The language of the specifications contained in this report corresponds exactly to SPECIAL as described in the reference manual.

SPECIAL allows for the definition of abstract data objects and operations performed upon these objects. The objects are represented as V-functions, i.e., functions that return a value. The collection of V-functions represents the state of the system being specified. Operations on objects are represented by O-functions. O-functions modify the values of V-functions and thereby change the state of the system. A third class of functions is that of OV-functions, i.e., functions that both change the system state and return a value.

A distinction among V-functions is that they can be either primitive or derived. The value of a derived V-function is simply an expression in terms of other V-functions. Only the values of primitive V-functions can be changed by O-functions and this implicitly changes the values of related derived

V-functions.

A second distinction among V-functions is that they can be either visible or hidden. A visible V-function is one that is available to the outside world. A hidden V-function can only be referenced from within other functions. By convention, the names of all hidden V-functions contained in the kernel specification begin with the prefix "h_".

An individual function specification comprises several different parts. For visible functions, the first part is always a list of exception conditions, i.e., conditions under which the function fails to operate or return a value. These exceptions apply only when a visible function is referenced from the outside world. When referenced internally, i.e., from within another function, exceptions are ignored. Similarly, hidden functions, which can only be referenced internally, have no exceptions. The remaining parts of a function specification differ for V-functions and O-functions. In the case of a V-function, the specification contains either an initial value for a primitive V-function or the derivation expression for a derived V-function. O-functions and OV-functions contain a list of "effects" that describe changes to V-functions. For an OV-function, the effects also define the output value. The order of effects within a given list is unimportant. All effects occur at once, i.e. instantaneously. Within an effects list, references are made to the old and new values of V-functions, i.e. the values before and after the effects have occurred. The new value of a V-function is indicated by a single quotation mark (') preceding the V-function name.

The entire set of functions which constitute the top-level specification are divided into groups called modules. The purpose of modules is to allow the specifier to conveniently organize a possibly large number of functions into small groupings that can be easily understood. The modularization serves no other purpose. Often, the modules may, in some ways, correspond to actual program modules in a perceived implementation. This, however, is not necessary.

Function references between modules are permitted. A function of one module can observe the values of V-functions, including hidden V-functions, of any other module. However, V-functions can only be directly modified by O- or OV-functions of the same module. Therefore, in order for one module to change a V-function of another module, the first module must invoke an O- or OV-function of the second module.

The specification of a module is divided into six sections as follows:

1. TYPES

This section defines new data types which supplement the primitive data types of the language.

2. DECLARATIONS

This section defines variable names and their associated data types.

3. PARAMETERS

This section defines named constants which, in SPECIAL, are called parameters.

4. DEFINITIONS

This section allows for the specification of macros.

5. EXTERNALREFS

This section defines all external functions referenced within the module.

6. FUNCTIONS

This section contains all of the individual function specifications for the module.

The Abstract Machine Interpreter

The purpose of the top-level specification is to define an abstract machine whose behavior represents that of a Multics security kernel as viewed by a user process. This abstract machine will be implemented as a combination of hardware and software, i.e., the Multics processor enhanced by a set of security kernel procedures. In this sense, the instruction set of the abstract machine is seen to be a combination of the actual hardware instructions of the Multics processor plus the "super-instructions" represented by callable entry points to the kernel procedures.

The specifications alone do not present a complete picture of the operation of an abstract machine. As mentioned earlier, they are concerned chiefly with the definition of abstract data objects and operations for examining and manipulating these objects. They do not, however, explain how these operations come to be executed. The answer to this question depends upon the concept of an abstract machine interpreter [BR1] [Rob]. This concept is briefly described below.

The abstract machine interpreter is a very general processor that has an associated instruction set as mentioned above. The abstract machine interpreter is cognizant of a set of processes, each of which is known to be either ready or blocked. For each instruction cycle, the abstract machine interpreter first selects a ready process. It then fetches the next instruction in the instruction stream for that process. This instruction is executed, i.e., the corresponding O-, V-, or OV-function is invoked. If an exception occurs, further action may be required by the abstract machine interpreter depending upon the particular instruction and the particular error involved.

Although this characterization of the abstract machine interpreter is both informal and incomplete, it hopefully provides some insight into the mechanism by which functions come to be executed. More details of the abstract machine interpreter will be revealed in the specification descriptions that follow.

It should be noted that functions are always executed in sequence, i.e. never in parallel. Hence, there is no concern over possible interference among functions. This lack of parallelism would be impractical, of course, in a real implementation. A technique has been described for allowing controlled parallel execution of functions without the danger of harmful interference [BR2]. This problem, however, is beyond the scope of this report.

CHAPTER IV

OVERVIEW OF THE MULTICS KERNEL SPECIFICATION

The Multics kernel specification is divided into the following modules:

clock
access_levels
processes
volumes
quota_cells
segments
address_spaces

As the module names suggest, each module takes the form of an object manager, i.e., each module defines the data representation and operations for a particular object type. These modules are separately described in the chapters that follow.

Taken together, the functions of the modules mentioned above define the top-level kernel interface. As described earlier, hidden V-functions contained within these modules are not visible at the kernel interface. Similarly, certain O- and OV-functions are also hidden from the kernel interface. This hiding is expressed by means of an interface specification. An interface specification identifies the modules that compose the interface and, for each module, identifies functions that are excluded from the interface.

The Multics kernel interface specification is shown below. Unlike the module specifications, the interface specification is not written in SPECIAL. A separate interface specification language (also developed at Stanford Research Institute) is used. The syntax of the interface specification is simply a list of module names enclosed in parentheses. The module names may optionally be followed by a "WITHOUT" clause that names functions which are explicitly excluded from the interface.

In the chapters that follow, reference is sometimes made to O- and OV-functions that are hidden from the kernel interface. All such functions are named in the "WITHOUT" clauses of the kernel interface specification.

```
(
INTERFACE Multics_kernel

(clock)

(access-levels)

(processes          WITHOUT   wake
                              send_signal)

(volumes)

(quota-cells        WITHOUT   set_quota
                              change_qc_refs
                              change_qc_pages_used)

(segments           WITHOUT   read_seg
                              write_seg
                              change_dtu_dtm)

(address-spaces     WITHOUT   revoke_vol_access
                              purge_address_space)
)
```

CHAPTER V

THE SYSTEM CLOCK AND UNIQUE IDENTIFIERS

The first module of the top-level specification describes the system clock. A function called "read_clock" is defined that returns the current clock value. This clock value can be incremented by a second function called "advance_clock" that is not available to ordinary processes. It is convenient to think of the clock mechanism as a special autonomous process whose only activity is to increment the clock. Note, however, that the clock cannot be incremented during the execution of other functions. The fundamental assumption that functions do not execute in parallel applies even to the ticking of the clock.

The clock provides a source of unique identifiers that are utilized by various components of the kernel. To ensure that unique identifiers are, indeed, unique, a record is kept of the last generated uid. A request for a new uid must wait until the clock advances to a new value.

```
MODULE  clock

     DECLARATIONS

INTEGER time;

     FUNCTIONS

VFUN read_clock() -> time;
     $(returns current clock reading)
     INITIALLY time = 0;

OFUN advance_clock();
     $(advances clock one time unit)
     EFFECTS
          'read_clock() = read_clock() + 1;

OVFUN get_uid() -> time;
     $(generates a new uid)
     DELAY UNTIL read_clock() > h_last_uid();
     EFFECTS
          time = read_clock();
          'h_last_uid() = time;

VFUN h_last_uid() -> time;
     $(returns last uid generated)
     HIDDEN;
     INITIALLY time = 0;


END_MODULE
```

CHAPTER VI

ACCESS LEVELS

The access_levels module is the only module of the kernel that understands the internal structure of an access level. As mentioned earlier, an access level is a combination of a security level and an integrity level. Each of these two levels has both a level number and a category set. Therefore, an access level is represented by a structure with four components.

The access_levels module contains functions for determining whether or not a process has read, write, or both read and write access to a given object. These functions implement the security and integrity access control policies of the model. The functions take as input a subject access level, an object access level, and a boolean value indicating whether the subject is trusted. All access control decisions made by the kernel depend on these functions.

In current Multics, certain processes are granted privileged access to various classes of objects. In the kernel-based Multics, privileges associated with object classes will not exist. Instead, the trusted process attribute is provided. A trusted process having the maximum security and integrity levels will have both read and write access to objects of all access levels.

```
MODULE  access_levels

     TYPES

level_number : {INTEGER ln | 0 <= ln AND ln <= max_ln};
category_set : {VECTOR OF BOOLEAN cs | LENGTH(cs) = cs_size};
access_level : STRUCT (level_number sln;
                       category_set scs;
                       level_number iln;
                       category_set ics);


     DECLARATIONS

access_level sub_al, ob_al;
INTEGER i;
BOOLEAN b, trusted;


     PARAMETERS

level_number max_ln $(maximum level number);
INTEGER cs_size $(category set size);


     FUNCTIONS

VFUN h_read_allowed(trusted; sub_al; ob_al) -> b;
     $(true if subject can read object)
     HIDDEN;
     DERIVATION sub_al.sln >= ob_al.sln AND
                (sub_al.iln <= ob_al.iln OR trusted) AND
                (FORALL i | i >= 1 AND i <= cs_size :
                (ob_al.scs[i] => sub_al.scs[i]) AND
                ((sub_al.ics[i] => ob_al.ics[i]) OR trusted));

VFUN h_write_allowed(trusted; sub_al; ob_al) -> b;
     $(true if subject can write object)
     HIDDEN;
     DERIVATION (sub_al.sln <= ob_al.sln OR trusted) AND
                sub_al.iln >= ob_al.iln AND
                (FORALL i | i >= 1 AND i <= cs_size :
                ((sub_al.scs[i] => ob_al.scs[i] ) OR trusted) AND
                (ob_al.ics[i] => sub_al.ics[i]));

VFUN h_read_write_allowed(trusted; sub_al; ob_al) -> b;
     $(true if subject can read and write object)
     HIDDEN;
     DERIVATION h_read_allowed(trusted, sub_al, ob_al) AND
                h_write_allowed(trusted, sub_al, ob_al);

END_MODULE
```

CHAPTER VII

PROCESSES

The processes module is concerned with the creation and deletion of processes, interprocess communication, and process timers.

At process creation, the access level of the new process and the trusted or untrusted status of the new process are specified. Use of the create_proc function is restricted to the initializer (trusted) process alone. For this reason, process creation cannot serve as an information path between untrusted processes. Therefore, knowledge of the existence of processes is not restricted.

The processes module supports the familiar block and wakeup primitives. The block function is of special interest to the abstract machine interpreter which keeps track of the state of each process, i.e., blocked or ready. In addition to block, a read_messages function is provided that simply reads any pending messages, but will not wait for a message to arrive.

The concept of event channels is unknown to the kernel. The kernel passes messages from one process to another without concern for event channel identifiers that may be contained in those messages. It is assumed that the supervisor will implement event channels.

The processes module also supports the signal mechanism. Functions are provided to send and receive signals. Note that the sending function, send_signal, is hidden from the kernel interface. Thus, a process cannot send a signal to another process. Signals are used by the kernel to permit immediate recognition of certain kernel-detected events such as quits and alarms. The masking of signals is handled outside of the kernel.

The key difference between signals and wakeups is that a signal is recognized immediately by the receiving process whereas a wakeup is not recognized until the receiving process next invokes the block or read_messages functions. This difference, however, is not evident from the specifications alone because it depends upon the operation of the abstract machine interpreter. It is assumed that the abstract machine interpreter is designed to check the signal flags for a process on each instruction cycle. If any of the flags is set, then the abstract machine interpreter causes a fault to occur, thereby giving the process an opportunity to use the receive_signal function. At the time a signal is sent, the abstract machine interpreter checks the state of the receiving process. If it is blocked, then a wakeup is also sent to the receiving process (with a dummy message that will be ignored). The purpose of the wakeup is simply to force

the receiving process into the ready state so that it can be forced to fault and notice a pending signal.

Another duty of the processes module is the support of real time alarms. When setting a real time alarm, the caller must specify the time of the alarm and also a wakeup message. If this message is zero, it indicates that the caller wishes to receive a signal at the time of the alarm. Otherwise, a wakeup is desired. The responsibility of watching the clock and sending the wakeup or signal rests with the abstract machine interpreter.

The present Multics supervisor supports the concept of process virtual time (sometimes called "cpu time"). In essence, process virtual time represents the real time that a process has actually been running on a processor with subtractions made for certain hidden events such as page faults and interrupts. The objective is to factor out time spent performing hidden operations that support the virtual process environment because this time is unpredictable and, in general, depends upon the activities of other processes. Taken to the limit, the concept of process virtual time would guarantee a fixed amount of time for the execution of each supervisor function. However, the current implementation of process virtual time is only an approximation to this limit.

Unfortunately, the current method of computing process virtual time cannot be specified without introducing knowledge of events such as page faults which are not otherwise visible in the top-level specification. If process virtual time were defined in an ideal fashion, i.e., if a fixed time value could be assigned to each function in the top-level specification, then a straightforward description would be possible. Failing this, however, no acceptable method of specification is apparent. Therefore, the concept of process virtual time does not appear in the top-level specification.

the receiving process into the ready state so that it can be forced to fault and notice a pending signal.

Another duty of the processes module is the support of real time alarms. When setting a real time alarm, the caller must specify the time of the alarm and also a wakeup message. If this message is zero, it indicates that the caller wishes to receive a signal at the time of the alarm. Otherwise, a wakeup is desired. The responsibility of watching the clock and sending the wakeup or signal rests with the abstract machine interpreter.

The present Multics supervisor supports the concept of process virtual time (sometimes called "cpu time"). In essence, process virtual time represents the real time that a process has actually been running on a processor with subtractions made for certain hidden events such as page faults and interrupts. The objective is to factor out time spent performing hidden operations that support the virtual process environment because this time is unpredictable and, in general, depends upon the activities of other processes. Taken to the limit, the concept of process virtual time would guarantee a fixed amount of time for the execution of each supervisor function. However, the current implementation of process virtual time is only an approximation to this limit.

Unfortunately, the current method of computing process virtual time cannot be specified without introducing knowledge of events such as page faults which are not otherwise visible in the top-level specification. If process virtual time were defined in an ideal fashion, i.e., if a fixed time value could be assigned to each function in the top-level specification, then a straightforward description would be possible. Failing this, however, no acceptable method of specification is apparent. Therefore, the concept of process virtual time does not appear in the top-level specification.

```
MODULE  processes

     TYPES

level_number : {INTEGER ln | 0 <= ln AND ln <= max_ln};
category_set : {VECTOR_OF BOOLEAN cs | LENGTH(cs) = cs_size};
access_level : STRUCT (level_number sln;
                       category_set scs;
                       level_number iln;
                       category_set ics);
process_uid : INTEGER;
message : INTEGER;
message_queue : VECTOR_OF message;


     DECLARATIONS

process_uid procuid, newproc, target;
access_level al;
message msg;
message_queue msg_queue;
BOOLEAN b;
INTEGER i, n;
INTEGER time, delta_t;


     PARAMETERS

process_uid initializer_id $(initializer process id);
INTEGER max_processes $(maximum number of processes);
INTEGER max_messages $(maximum size of process message queue),
        nsignals $(number of different signals),
        real_timer_signal $(signal code for real timer runout);


     DEFINITIONS

INTEGER first_signal(procuid) IS
     MIN({i | h_signal_flag(procuid, i)});

BOOLEAN no_process(procuid) IS
     ~h_proc_exists(procuid);

BOOLEAN write_not_allowed(procuid; al) IS
     ~h_write_allowed(h_proc_trusted(procuid), h_proc_al(procuid), al);


     EXTERNALREFS

FROM clock:
     VFUN read_clock() -> time;
```

```
     OVFUN get_uid() -> procuid;

FROM access_levels:
     level_number max_ln $(maximum level number);
     INTEGER cs_size $(category set size);
     VFUN h_write_allowed(b; al; al) -> b;

FROM address_spaces:
     OFUN purge_address_space(procuid);


     FUNCTIONS

OVFUN create_proc(al; b)[procuid] -> newproc;
     $(creates a new process)
     EXCEPTIONS
          procuid ~= initializer_id;
          h_proc_count() = max_processes;
     EFFECTS
          newproc = EFFECTS_OF get_uid();
          'h_proc_exists(newproc) = TRUE;
          'h_proc_count() = h_proc_count() +1;
          'h_proc_al(newproc) = al;
          'h_proc_trusted(newproc) = b;

VFUN h_proc_exists(procuid) -> b;
     $(true if process exists)
     HIDDEN;
     INITIALLY b = FALSE;

VFUN h_proc_count() -> n;
     $(number of processes)
     HIDDEN;
     INITIALLY n = 0;

VFUN h_proc_al(procuid) -> al;
     $(returns access level of process)
     INITIALLY al = ?;

VFUN proc_al()[procuid] -> al;
     $(external form of h_proc_al)
     DERIVATION h_proc_al(procuid);

VFUN h_proc_trusted(procuid) -> b;
     $(true if process is trusted)
     INITIALLY b = ?;

VFUN proc_trusted()[procuid] -> b;
     $(external form of h_proc_trusted)
     DERIVATION h_proc_trusted(procuid);

OFUN delete_proc(target)[procuid];
     $(deletes a process)
```

```
     EXCEPTIONS
          procuid ~= initializer_id;
          no_process(target);
     EFFECTS
          'n_proc_count() = h_proc_count() - 1;
          'h_proc_exists(target) = FALSE;
          EFFECTS_OF purge_address_space(target);

OFUN wake(target; msg);
     $(transmits a message to target process
        if target queue is full, message is lost)
     DEFINITIONS
          INTEGER n IS LENGTH(h_proc_msg_queue(target));
          INTEGER m IS IF n = max_messages THEN n ELSE n+1;
     EFFECTS
          h_proc_msg_queue(target) = VECTOR (FOR i FROM 1 TO m:
          IF i <= n THEN h_proc_msg_queue(target)[i] ELSE msg);

OFUN wakeup(target; msg)[procuid];
     $(external form of wake)
     EXCEPTIONS
          no_process(target);
          write_not_allowed(procuid, h_proc_al(target));
     EFFECTS
          EFFECTS_OF wake(target, msg);

OVFUN read_messages()[procuid] -> msg_queue;
     $(empties message queue for process)
     EFFECTS
          msg_queue = h_proc_msg_queue(procuid);
          'h_proc_msg_queue(procuid) = VECTOR ();

OVFUN block()[procuid] -> msg_queue;
     $(empties message queue for process
        if queue is already empty, waits for message to arrive)
     DELAY UNTIL LENGTH(h_proc_msg_queue(procuid)) > 0;
     EFFECTS
          EFFECTS_OF read_messages(procuid) = msg_queue;

VFUN h_proc_msg_queue(procuid) -> msg_queue;
     $(returns message queue for process)
     HIDDEN;
     INITIALLY msg_queue = VECTOR ();

OFUN send_signal(target; n);
     $(sends signal n to target process)
     EFFECTS
          'h_signal_flag(target, n) = TRUE;

OVFUN receive_signal()[procuid] -> n;
     $(receives lowest number signal)
     EFFECTS
          n = first_signal(procuid);
```

```
          n > 0 => 'h_signal_flag(procuid, n) = FALSE;

VFUN h_signal_flag(procuid; n) -> b;
     $(true if signal n sent to process)
     HIDDEN;
     INITIALLY b = FALSE;

OFUN set_real_timer(delta_t; msg)[procuid];
     $(sets real timer alarm for process)
     EFFECTS
          'h_real_timer(procuid) = (IF delta_t <= 0 THEN ?
                                    ELSE read_clock() + delta_t);
          delta_t > 0 => 'h_real_timer_msg(procuid) = msg;

VFUN h_real_timer(procuid) -> time;
     $(returns time of next real timer alarm)
     HIDDEN;
     INITIALLY time = ?;

VFUN h_real_timer_msg(procuid) -> msg;
     $(returns real timer wakeup message)
     HIDDEN;
     INITIALLY msg = ?;


END_MODULE
```

CHAPTER VIII

VOLUMES

A volume is a logical subdivision of secondary storage. A single volume may comprise one or more physical storage devices, i.e., disk packs, although this fact is largely concealed outside the kernel. The volumes module is concerned only with volume creation and deletion and volume mounting. The control of volume storage space is handled by other modules.

At the time a volume is created (i.e., registered), minimum and maximum access levels for the volume are specified. These access levels delimit the range of access levels of information that can be stored on the volume. The minimum access level of the volume also serves as a visibility access level for the volume. The protection of information stored on a volume, however, does not depend on these access levels. Each object that resides on a volume must have its own access level. The purpose of the volume access levels is, for the most part, to satisfy certain operational requirements. For example, highly sensitive information can be segregated onto specific disk packs that receive special handling.

Also, at volume creation time, an initial quota cell is created for a volume. The quota value of the quota cell is set to the volume size, i.e. the number of pages on the volume. The role of the initial quota cell is explained in the description of the quota_cells module.

The specifications place no limit on the number of volumes that can be created. In practice, however, the kernel would need to maintain a table of volumes. If any process can create a volume, the table becomes an instance of the shared finite resource problem. A simple solution would be to restrict the creation of volumes to trusted processes.

The only operations that can be performed on a volume are mounting and demounting. Both of these operations require that the user's access level be equal to the volume minimum access level. This is necessary because the mounting and demounting of a volume can be detected by any process at an access level equal to or greater than the volume minimum access level.

Unfortunately, volume mounting produces another instance of the shared finite resource problem. In this case, the finite resource is the collection of disk drives. The mounting and demounting of volumes could be restricted to trusted processes. This, however, seems clumsy and undesirable. Another possibility is to assign access levels to the disk

drives. The kernel could then require that for a process to mount a volume on a disk drive (or set of drives) the process access level, the volume minimum access level, and the drive access level must all be equal. The access levels of disk drives could be changed dynamically, if necessary, by a trusted process presumably under operator control.

After mounting a volume, a user might find it necessary to destroy his current process and create a new process at a greater access level in order to use information on the volume having an access level greater than the volume minimum. Later, the user must again create a new process at the volume minimum access level in order to issue a demount request. This inconvenience is, of course, due to the fact that volumes are multiple access level objects. If volumes were restricted to a single access level, this problem would vanish. However, a requirement to dedicate a whole volume (at least one full disk pack) to a single access level might be economically undesirable. In any event, single access level volumes are simply a degenerate case of multiple access level volumes. Therefore, it is always possible to create single access level volumes if desired.

The above discussion of volume mounting is concerned with the actual physical mounting of volumes. The current Multics user interface, however, supports the concept of volume attachment. A process must attach a volume before using it. The first process to attach a volume causes it to be mounted. Similarly, the last process to detach a volume causes it to be demounted. This scheme cannot be entirely supported because of the previously explained need for a user to change access levels (and hence processes) during the time that a volume is mounted.

```
MODULE  volumes

     TYPES

level_number : {INTEGER ln | 0 <= ln AND ln <= max_ln};
category_set : {VECTOR OF BOOLEAN cs | LENGTH(cs) = cs_size};
access_level : STRUCT (level_number sln;
                       category_set scs;
                       level_number iln;
                       category_set ics);
process_uid : INTEGER;
volume_uid : INTEGER;
quota_cell_uid : INTEGER;


     DECLARATIONS

volume_uid voluid;
process_uid procuid;
quota_cell_uid qcuid;
access_level min_al, max_al, al;
BOOLEAN b;
INTEGER i;


     PARAMETERS

INTEGER volume_size $(number of pages on a volume);


     DEFINITIONS

BOOLEAN unordered_access_levels(min_al; max_al) IS
     ~h_write_allowed(FALSE, min_al, max_al);

BOOLEAN write_not_allowed(procuid; al) IS
     ~h_write_allowed(h_proc_trusted(procuid), h_proc_al(procuid), al);

BOOLEAN no_volume(procuid; voluid) IS
     IF ~h_vol_exists(voluid) THEN TRUE
     ELSE ~h_read_allowed(h_proc_trusted(procuid), h_proc_al(procuid),
                          h_vol_min_al(voluid));

BOOLEAN mounted_volume(voluid) IS
     h_vol_mounted(voluid);

BOOLEAN unmounted_volume(procuid; voluid) IS
     IF ~h_vol_mounted(voluid) THEN TRUE
     ELSE ~h_read_allowed(h_proc_trusted(procuid), h_proc_al(procuid),
                          h_vol_min_al(voluid));
```

```
     EXTERNALREFS

FROM clock:
     OVFUN get_uid() -> voluid;

FROM access_levels:
     level_number max_ln $(maximum level number);
     INTEGER cs_size $(category set size);
     VFUN n_write_allowed(b; al; al) -> b;
     VFUN n_read_allowed(b; al; al) -> b;

FROM processes:
     VFUN h_proc_al(procuid) -> al;
     VFUN h_proc_trusted(procuid) -> b;

FROM quota_cells:
     OVFUN create_quota_cell(voluid; al; al)[procuid] -> qcuid;
     OFUN set_quota(qcuid; i);

FROM address_spaces:
     OFUN revoke_vol_access(voluid; procuid);


     FUNCTIONS

OVFUN create_volume(min_al; max_al)[procuid] -> voluid;
     $(creates a new volume)
     DEFINITIONS
          quota_cell_uid qcuid IS 'n_vol_init_qc(voluid);
     EXCEPTIONS
          unordered_access_levels(min_al, max_al);
          write_not_allowed(procuid, min_al);
     EFFECTS
          voluid = EFFECTS_OF get_uid();
          'n_vol_exists(voluid) = TRUE;
          'n_vol_min_al(voluid) = min_al;
          'n_vol_max_al(voluid) = max_al;
          qcuid = EFFECTS_OF create_quota_cell(voluid, min_al, min_al,
                                               procuid);
          EFFECTS_OF set_quota(qcuid, volume_size);

VFUN n_vol_exists(voluid) -> b;
     $(true if volume exists)
     HIDDEN;
     INITIALLY b = FALSE;

VFUN n_vol_min_al(voluid) -> min_al;
     $(returns minimum access level of volume)
     HIDDEN;
     INITIALLY min_al = ?;

VFUN vol_min_al(voluid)[procuid] -> min_al;
```

```
     $(external form of h_vol_min_al)
     EXCEPTIONS
          no_volume(procuid, voluid);
     DERIVATION h_vol_min_al(voluid);

VFUN h_vol_max_al(voluid) -> max_al;
     $(returns maximum access level of volume)
     HIDDEN;
     INITIALLY max_al = ?;

VFUN vol_max_al(voluid)[procuid] -> max_al;
     $(external form of h_vol_max_al)
     EXCEPTIONS
          no_volume(procuid, voluid);
     DERIVATION h_vol_max_al(voluid);

VFUN h_vol_init_qc(voluid) -> qcuid;
     $(returns initial quota cell uid for volume)
     HIDDEN;
     INITIALLY qcuid = ?;

VFUN vol_init_qc(voluid)[procuid] -> qcuid;
     $(external form of h_vol_init_qc)
     EXCEPTIONS
          no_volume(procuid, voluid);
     DERIVATION h_vol_init_qc(voluid);

OPUN delete_volume(voluid)[procuid];
     $(deletes a volume)
     EXCEPTIONS
          no_volume(procuid, voluid);
          mounted_volume(voluid);
          write_not_allowed(procuid, h_vol_min_al(voluid));
     EFFECTS
          'h_vol_exists(voluid) = FALSE;

OPUN mount_volume(voluid)[procuid];
     $(mounts a volume)
     EXCEPTIONS
          no_volume(procuid, voluid);
          mounted_volume(voluid);
          write_not_allowed(procuid, h_vol_min_al(voluid));
     EFFECTS
          'h_vol_mounted(voluid) = TRUE;

VFUN h_vol_mounted(voluid) -> b;
     $(true if volume is mounted)
     HIDDEN;
     INITIALLY b = FALSE;

OPUN demount_volume(voluid)[procuid];
     $(demounts a volume)
     EXCEPTIONS
```

```
        unmounted_volume(procuid, voluid);
        write_not_allowed(procuid, h_vol_min_al(voluid));
    EFFECTS
        'n_vol_mounted(voluid) = FALSE;
        EFFECTS_OF revoke_vol_access(voluid, procuid);


END_MODULE
```

CHAPTER IX

QUOTA CELLS

Quota cells are the mechanism by which the sharing of pages on a volume is controlled. Each segment, and hence each page of each segment, is assigned to a particular quota cell. Having no knowledge of directories, the kernel presents a general interface that would allow any arbitrary collection of segments having the same access level to be assigned to the same quota cell. The supervisor, however, will impose an arrangement of quota cells and segments that reflects the directory hierarchy. Thus, the combined operation of the supervisor and kernel will produce a quota mechanism essentially equivalent to that of current Multics.

At the time a quota cell is created, a volume, an access level, and a visibility access level are specified. The two access levels must be within the volume range. In the interest of making volumes self-describing (and thereby invulnerable to the failures of other volumes), it is assumed that all quota cells for a volume are stored on the volume itself.

Storage space on a volume is first made available by the creation of an initial quota cell as previously described. Both the access level and the visibility access level of an initial quota cell must equal the volume minimum access level. As described below, this permits the initial quota to be distributed to higher access levels as needed.

Aside from the initial quota cell, all other quota cells are created with a quota of zero pages. In order to assign a non-zero quota to a new quota cell, quota must be moved from some other quota cell on the same volume. With regard to access levels, quota cannot be moved downward, i.e., the move_quota operation insists that the target quota cell have an equal or greater access level than the source quota cell. Only trusted processes can move quota downward.

A reference count is associated with each quota cell. This count is the number of segments that charge pages to the quota cell. So long as this count is non-zero, the quota cell cannot be deleted.

Another attribute of a quota cell is the count of pages used. This count is changed by the creation and deletion of pages in associated segments as specified in the segments module. For all quota cells, the pages used count is not permitted to exceed the quota.

The quota cells module is also responsible for metering the use of pages for accounting purposes. To this end, the pages used count is integrated over time to produce a "time-record product" (trp). Each time the pages used count is changed, the trp must be updated. If a quota cell is deleted, its trp must be added to that of some other quota cell on the same volume so that prior usage of the deleted quota cell will be accounted for. A function is also provided to read and reset to zero the trp of a quota cell. This function is needed by the accounting system to collect charges for one billing cycle and, at the same time, begin a new cycle.

As described above, the quota cell mechanism controls the sharing of pages among different access levels and thus represents a solution to the shared finite resource problem for pages. This solution depends, however, on the assumption that quota cannot be over-allocated as is current practice at some Multics sites. If this were permitted, i.e., if the quota for a volume could be set higher than the actual number of available pages, then the exhaustion of pages on a volume could occur before the exhaustion of quota. This would constitute an uncontrolled information path. Of course, on single access level volumes, this information path is of no consequence. Hence, it would be possible to allow over-allocation of quota on such volumes.

No limit is placed on the number of quota cells that can be created on a volume. Ironically, in attempting to solve one instance of the shared finite resource problem, a new instance is created. Some method for controlling the number of quota cells on a volume is needed.

The proposed quota cell mechanism yields two incompatibilities with the current Multics quota scheme. First, it will not be possible to convert a directory from terminal quota status to non-terminal quota status or vice versa. Second, a "pages used" value will not be maintained for non-terminal quota directories.

Both of the above incompatibilities stem from a single simplification of the quota cell mechanism. In current Multics, there exists a class of "indirect" quota cells. Specifically, these are the quota cells associated with non-terminal quota directories. In order to determine whether a page can be added to a segment in such a directory, it is necessary to follow a chain of indirect quota cells until reaching a quota cell associated with a terminal quota directory. In the proposed kernel quota mechanism, there are no indirect quota cells. This eliminates a significant amount of complexity and, at the same time, produces the two aforementioned incompatibilities.

The impact of these incompatibilities on users can be estimated. The restriction on converting directories between terminal and non-terminal quota status should have little or no effect on the vast majority of users. At single access level sites, quota will rarely be moved below the user home directory level. Statistics gathered on the MIT system support this claim [Jan]. At multiple access level sites, quota will be moved below the user home directory level to accomodate upgraded directories. In current Multics, upgraded directories are required to have a quota at all times and thus already obey the proposed restriction. The elimination of a pages used value for non-terminal quota directories may have a more serious impact. At certain times, this feature could be valuable to any user. Of course, it is always possible to calculate the pages used of a subtree (with some dynamic uncertainty) by summing the pages used of every segment in the subtree. This could be an expensive operation. However, if done infrequently, the cost may still be less than the constant overhead associated with maintaining indirect quota cells. Remember that indirect quota cells waste wired memory and require extra computation at page fault time. Perhaps some insight could be gained by metering the frequency of requests for the pages used value of indirect quota cells in current Multics.

Two other aspects of the way the supervisor is expected to use quota cells deserve mention. First, the pages of a terminal quota directory will be charged to the quota cell associated with that directory, not to the quota cell of a parent directory. This is necessary to meet security requirements for an upgraded directory. Second, the supervisor may permit quotas to be assigned to individual non-directory segments. This would allow the creation of upgraded segments, a feature not found in current Mutlics.

```
MODULE  quota_cells

     TYPES

level_number : {INTEGER ln | 0 <= ln AND ln <= max_ln};
category_set : {VECTOR OF BOOLEAN cs | LENGTH(cs) = cs_size};
access_level : STRUCT (level_number sln;
                       category_set scs;
                       level_number iln;
                       category_set ics);
process_uid : INTEGER;
volume_uid : INTEGER;
quota_cell_uid : INTEGER;

     DECLARATIONS

process_uid procuid;
volume_uid voluid;
quota_cell_uid qcuid, from_qcuid, to_qcuid;
access_level al, val;
INTEGER npages, n;
INTEGER time, trp;
BOOLEAN b;

     DEFINITIONS

BOOLEAN unmounted_volume(procuid; voluid) IS
     IF ~h_vol_mounted(voluid) THEN TRUE
     ELSE ~h_read_allowed(h_proc_trusted(procuid), h_proc_al(procuid),
                          h_vol_min_al(voluid));

BOOLEAN outside_vol_levels(voluid; val; al) IS
     ~h_write_allowed(FALSE, h_vol_min_al(voluid), val) OR
     ~h_write_allowed(FALSE, al, h_vol_max_al(voluid));

BOOLEAN unordered_access_levels(val; al) IS
     ~h_write_allowed(FALSE, val, al);

BOOLEAN write_not_allowed(procuid; al) IS
     ~h_write_allowed(h_proc_trusted(procuid), h_proc_al(procuid), al);

BOOLEAN no_quota_cell(procuid; voluid; qcuid) IS
     IF ~h_qc_exists(voluid, qcuid) THEN TRUE
     ELSE ~h_read_allowed(h_proc_trusted(procuid), h_proc_al(procuid),
                          h_qc_visibility_al(qcuid));

BOOLEAN read_not_allowed(procuid; al) IS
     ~h_read_allowed(h_proc_trusted(procuid), h_proc_al(procuid), al);

BOOLEAN read_write_not_allowed(procuid; al) IS
```

```
     'h read write allowed(h proc trusted(procuid), h proc al(procuid),
                           al);

BOOLEAN non zero quota(qcuid) IS
     h qc pages(qcuid) ~= 0;

BOOLEAN non zero refs(qcuid) IS
     h_qc_refs(qcuid) ~= 0;

BOOLEAN invalid quota_change(npages) IS
     npages < 0;

BOOLEAN insufficient_quota(qcuid; npages) IS
     h_qc_pages(qcuid) - h_qc_pages_used(qcuid) < npages;


     EXTERNALREFS

FROM clock:
     VFUN read clock() -> time;
     OVFUN get uid() -> qcuid;

FROM access levels:
     level number max_ln $(maximum level number);
     INTEGER cs size $(category set size);
     VFUN h read allowed(b; al; al) -> b;
     VFUN h_write_allowed(b; al; al) -> b;
     VFUN h_read_write_allowed(b; al; al) -> b;

FROM processes:
     VFUN h proc al(procuid) -> al;
     VFUN h proc trusted(procuid) -> b;

FROM volumes:
     VFUN h_vol_mounted(voluid) -> b;
     VFUN h_vol_min al(voluid) -> al;
     VFUN h vol max_al(voluid) -> al;


     FUNCTIONS

OVFUN create_quota_cell(voluid; val; al)[procuid] -> qcuid;
     $(creates a new quota cell)
     EXCEPTIONS
          unordered access levels(val, al);
          write_not_allowed(procuid, val);
          unmounted volume(procuid, voluid);
          outside_vol_levels(voluid, val, al);
     EFFECTS
          qcuid = EFFECTS_OF get_uid();
          'h_qc_al(qcuid) = al;
          'h_qc_visibility_al(qcuid) = val;
```

```
          'h_qc_exists(voluid, qcuid) = TRUE;

VFUN h_qc_exists(voluid; qcuid) -> b;
     $(true if quota cell exists)
     HIDDEN;
     INITIALLY b = FALSE;

VFUN h_qc_visibility_al(qcuid) -> al;
     $(returns access level of quota cell visibility)
     HIDDEN;
     INITIALLY al = ?;

VFUN qc_visibility_al(voluid; qcuid)[procuid] -> al;
     $(external form of h_qc_visibility_al)
     EXCEPTIONS
          unmounted_volume(procuid, voluid);
          no_quota_cell(procuid, voluid, qcuid);
     DERIVATION h_qc_visibility_al(qcuid);

VFUN h_qc_al(qcuid) -> al;
     $(returns access level of quota cell)
     HIDDEN;
     INITIALLY al = ?;

VFUN qc_al(voluid; qcuid)[procuid] -> al;
     $(external form of h_qc_al)
     EXCEPTIONS
          unmounted_volume(procuid, voluid);
          no_quota_cell(procuid, voluid, qcuid);
     DERIVATION h_qc_al(qcuid);

OPUN set_quota(qcuid; npages);
     $(sets quota of quota cell
       used only to initialize first quota cell on a volume)
     EFFECTS
          'h_qc_pages(qcuid) = npages;

OPUN move_quota(voluid; from_qcuid; to_qcuid; npages)[procuid];
     $(moves page quota from one cell to another)
     EXCEPTIONS
          invalid_quota_change(npages);
          unmounted_volume(procuid, voluid);
          no_quota_cell(procuid, voluid, from_qcuid);
          no_quota_cell(procuid, voluid, to_qcuid);
          read_write_not_allowed(procuid, h_qc_al(from_qcuid));
          write_not_allowed(procuid, h_qc_al(to_qcuid));
          insufficient_quota(from_qcuid, npages);
     EFFECTS
          'h_qc_pages(from_qcuid) = h_qc_pages(from_qcuid) - npages;
          'h_qc_pages(to_qcuid) = h_qc_pages(to_qcuid) + npages;

VFUN h_qc_pages(qcuid) -> npages;
     $(returns page quota for quota cell)
```

```
     HIDDEN;
     INITIALLY npages = 0;

VFUN qc_pages(voluid; qcuid)[procuid] -> npages;
     $(external form of h_qc_pages)
     EXCEPTIONS
          unmounted_volume(procuid, voluid);
          no_quota_cell(procuid, voluid, qcuid);
          read_not_allowed(procuid, h_qc_al(qcuid));
     DERIVATION h_qc_pages(qcuid);

VFUN h_qc_refs(qcuid) -> n;
     $(returns quota cell reference count)
     HIDDEN;
     INITIALLY n = 0;

VFUN qc_refs(voluid; qcuid)[procuid] -> n;
     $(external form of h_qc_refs)
     EXCEPTIONS
          unmounted_volume(procuid, voluid);
          no_quota_cell(procuid, voluid, qcuid);
          read_not_allowed(procuid, h_qc_al(qcuid));
     DERIVATION h_qc_refs(qcuid);

OFUN change_qc_refs(qcuid; n);
     $(changes quota cell reference count)
     EFFECTS
          'h_qc_refs(qcuid) = h_qc_refs(qcuid) + n;

OFUN delete_quota_cell(voluid; qcuid; to_qcuid)[procuid];
     $(deletes a quota cell)
     EXCEPTIONS
          unmounted_volume(procuid, voluid);
          no_quota_cell(procuid, voluid, qcuid);
          no_quota_cell(procuid, voluid, to_qcuid);
          write_not_allowed(procuid, h_qc_visibility_al(qcuid));
          read_not_allowed(procuid, h_qc_al(qcuid));
          write_not_allowed(procuid, h_qc_al(to_qcuid));
          non_zero_quota(qcuid);
          non_zero_refs(qcuid);
     EFFECTS
          'h_qc_trp(to_qcuid) = h_qc_trp(to_qcuid) + h_qc_trp(qcuid);
          'h_qc_exists(voluid, qcuid) = FALSE;

VFUN h_qc_pages_used(qcuid) -> npages;
     $(returns pages used for quota cell)
     HIDDEN;
     INITIALLY npages = 0;

VFUN qc_pages_used(voluid; qcuid)[procuid] -> npages;
     $(external form of h_qc_pages_used)
     EXCEPTIONS
          unmounted_volume(procuid, voluid);
```

```
          no_quota_cell(procuid, voluid, qcuid);
          read_not_allowed(procuid, h_qc_al(qcuid));
     DERIVATION h_qc_pages_used(qcuid);

OPUN change_qc_pages_used(qcuid; npages);
     $(changes pages used for quota cell)
     DEFINITIONS
          INTEGER time IS 'h_qc_trp_updated(qcuid)
                          - h_qc_trp_updated(qcuid);
     EFFECTS
          'h_qc_pages_used(qcuid) = h_qc_pages_used(qcuid) + npages;
          'h_qc_trp_updated(qcuid) = read_clock();
          'h_qc_trp(qcuid) = h_qc_trp(qcuid) +
                             (time * h_qc_pages_used(qcuid));

VFUN h_qc_trp(qcuid) -> trp;
     $(returns time record product of quota cell)
     HIDDEN;
     INITIALLY trp = 0;

VFUN h_qc_trp_updated(qcuid) -> time;
     $(returns time trp last updated)
     HIDDEN;
     INITIALLY time = 0;

VFUN qc_trp(voluid; qcuid)[procuid] -> trp;
     $(external form of h_qc_trp)
     DEFINITIONS
          INTEGER time IS read_clock() - h_qc_trp_updated(qcuid);
     EXCEPTIONS
          unmounted_volume(procuid, voluid);
          no_quota_cell(procuid, voluid, qcuid);
          read_not_allowed(procuid, h_qc_al(qcuid));
     DERIVATION h_qc_trp(qcuid) + (time * h_qc_pages_used(qcuid));

OVFUN reset_trp(voluid; qcuid)[procuid] -> trp;
     $(reads and resets trp for new accounting period)
     DEFINITIONS
          INTEGER time IS read_clock() - h_qc_trp_updated(qcuid);
     EXCEPTIONS
          unmounted_volume(procuid, voluid);
          no_quota_cell(procuid, voluid, qcuid);
          read_write_not_allowed(procuid, h_qc_al(qcuid));
     EFFECTS
          trp = h_qc_trp(qcuid) + (time * h_qc_pages_used(qcuid));
          'h_qc_trp(qcuid) = 0;
          'h_qc_trp_updated(qcuid) = read_clock();


END_MODULE
```

CHAPTER X

SEGMENTS

The segments module is, of course, responsible for the management of segments, the logical storage units of Multics. Within this module, segments are identified by uid only. The local process view of segments, where segment numbers are used, is defined by the address_spaces module described later.

At segment creation, a visibility access level, a volume, and a quota cell must be specified. The new segment resides on the specified volume and charges its pages to the specified quota cell (which must also reside on the same volume). The access level of the new segment is, by necessity, equal to that of its associated quota cell.

No limit is placed on the number of segments that can be created on a given volume. In practice, however, a volume table of contents (VTOC) entry will be required for each segment. Thus, the VTOC represents yet another instance of the shared finite resource problem. At present, Multics provides no direct means for controlling the consumption of VTOC space. Limits on the size of directories provide an indirect form of control, but this is entirely inadequate. Even if security considerations are ignored, there still exists a serious problem in that a single user cannot be prevented from consuming an arbitrarily large number of VTOC entries.

One possible solution to this problem is to introduce the concept of segment quotas. The proposed quota cell mechanism already maintains a segment reference count. By enforcing limits on these segment counts, the use of VTOC entries could be controlled. This implies, however, an additional burden on the user. When moving quota, the user would need to specify not only the amount of page quota to move but also the amount of segment quota.

Functions are defined for reading and writing the contents of segments. These functions have certain side-effects, as well. Initially, all words of a segment have a value of zero.

The segments module maintains a page map (i.e. a record of existing pages) for each segment. Initially, a segment has no existing pages. A page is created the first time a word of the page is written. The page remains in existence until the segment is truncated to a size that excludes the page.

The treatment of zero pages differs from that of current Multics in two respects. First, reading from a zero page does

not cause the page to be created. Second, zero pages are not automatically deleted at the time they become candidates for removal from main memory.

The first difference is motivated by security considerations. The page map for a segment has the same access level as the segment itself. Therefore, a user having read access to a segment, but not write access, cannot be permitted to modify the page map. Hence, reading a nonexistent page cannot be allowed to cause the page to be created. Unfortunately, this restriction causes a difficult problem. The solution appears to require a change to the Multics processor.

One proposal suggests that page table words contain a fault-on-write bit that can be used to detect the first time a page is written [SBS]. All page table words for zero pages could then be directed to a single zero page. Only when an attempt is made to write a zero page would a new page be created.

This idea can be carried a step further by having a "zero-page" bit instead of a fault-on-write bit in each page table word. The zero-page bit would imply that any attempt to read from the page should always return a zero without any fault or memory reference. An attempt to write the page would cause a page fault, as usual.

Hardware changes, of course, are to be avoided wherever possible. In this case, a hardware change seems unavoidable. However, an interim solution is possible that avoids modifying the processor. The effect of the fault-on-write bit could be simulated by a special memory box. This memory box would have no real storage. It would simply return a zero for each read operation and would cause a fault on each write operation. The kernel would arrange for the page table words of zero pages to always address this special memory box. Dedicating a memory port to this special memory box essentially wastes a portion of the available absolute address space. This, however, does not appear to be a serious drawback since no Multics system yet contemplated requires the maximum amount of main memory.

The second difference in the treatment of zero pages, i.e., the elimination of automatic deletion of zero pages, is not motivated by security considerations. Rather, this difference is motivated by a desire to avoid introducing hidden mechanisms into the kernel top-level specification. In current Multics, the deletion of zero pages occurs at the time a page is selected for removal from main memory. This time cannot be predicted using only knowledge available at the user interface and, consequently, gives rise to non-deterministic and sometimes anomalous behavior. Such behavior depends upon factors such as the size of main memory which are not visible

in the top-level specification.

The consequences of eliminating automatic deletion of zero pages seem tolerable, if not desirable. This change is not incompatible; it should not cause existing programs to stop working. It will, however, cause storage charges to be higher for segments maintained by programs that deliberately or inadvertently accomplish page deletion through zeroing rather than truncation. Hopefully, the number of such segments is small and the programs that maintain them can be changed eventually. At the same time, this change has the advantage of making it possible to determine the size of a segment with certainty. This is not now the case due to the previously described non-deterministic behavior of the paging mechanism.

Two other segment attributes are the date-time used (dtu) and the date-time modified (dtm). For reasons described below, these two attributes are maintained in a different fashion and, consequently, have somewhat different meanings in the kernel-based Multics.

In current Multics, the dtu and dtm for a segment are updated each time the VTOC entry for the segment is updated. This occurs when a segment is deactivated. It also occurs when, in searching the active segment table for a segment to deactivate, a segment is found with a page table that has been modified. This method of updating relies on hidden mechanisms and, from the user's viewpoint, results in non-deterministic and often anomalous behavior. Such behavior depends upon factors which are not visible in the top-level specification such as the size of the active segment table.

In order to avoid introducing hidden mechanisms into the top-level kernel specification, a new method of updating dtu and dtm is proposed. Ideally, dtu and dtm should be updated on each segment reference. Since this is not possible, however, the kernel maintains two indicators called the "used" and "modified" flags. A function is provided to update dtu and dtm based on the settings of the used and modified flags. As will be seen in the address spaces module, this function is invoked automatically at the time a segment is terminated. It is also invoked automatically for all initiated segments at the time a process is terminated. Where greater accuracy is required, the updating function can be invoked explicitly. This would allow an editor, for example, to update dtu and dtm after writing out a buffer.

A request to update dtu and dtm is honored only if the corresponding used and modified flags have been turned on. The modified flag is set each time a segment is written. In principle, the used flag should be set each time a segment is read or written. Unfortunately, security considerations prohibit this interpretation of the used flag. The access

level of the used flag is equal to the access level of the segment. Therefore, just as in the case of the page map, a process with only read access to the segment cannot be permitted to change the used flag.

In order to provide at least some support for the dtu attribute, and yet not violate security, the kernel offers a compromise. The used flag is always set by write operations on a segment. For read operations, the used flag is only set if the process access level equals the segment access level. Under these conditions, there is no violation of security. Thus, the dtu should be interpreted as the last time a segment was used by a process of the same access level. Note that in single access level systems, this difference causes no change to the meaning of dtu.

```
MODULE segments

    TYPES

level_number : {INTEGER ln | 0 <= ln AND ln <= max_ln};
category_set : {VECTOR OF BOOLEAN cs | LENGTH(cs) = cs_size};
access_level : STRUCT (level_number sln;
                       category_set scs;
                       level_number iln;
                       category_set ics);
process_uid : INTEGER;
volume_uid : INTEGER;
quota_cell_uid : INTEGER;
segment_uid : INTEGER;
segment_offset : {INTEGER so | 0 <= so AND so <= max_offset};
machine_word : INTEGER;
page_number : {INTEGER pn | 1 <= pn AND pn <= (max_offset+1)/page_size};

    DECLARATIONS

process_uid procuid;
volume_uid voluid;
quota_cell_uid qcuid;
segment_uid seguid;
access_level al, val;
segment_offset offset;
machine_word word;
page_number pageno;
INTEGER time;
INTEGER i;
VECTOR OF INTEGER tt;
BOOLEAN b;
VECTOR OF BOOLEAN bmap;

    PARAMETERS

segment_offset max_offset $(maximum segment offset);
page_number max_pageno $(maximum page number);
INTEGER page_size $(number of words in a page);

    DEFINITIONS

INTEGER total_pages(seguid; pageno) IS
    CARDINALITY({i | i >= pageno AND i <= max_pageno AND
                 n_page_exists(seguid, i)});

BOOLEAN unmounted_volume(procuid; voluid) IS
    IF ~n_vol_mounted(voluid) THEN TRUE
    ELSE ~n_read_allowed(n_proc_trusted(procuid), n_proc_al(procuid),
```

```
                        n_vol_min_al(voluid));

BOOLEAN no_quota_cell(procuid; voluid; qcuid) IS
     IF ~n_qc_exists(voluid, qcuid) THEN TRUE
     ELSE ~n_read_allowed(n_proc_trusted(procuid), n_proc_al(procuid),
                        n_qc_visibility_al(qcuid));

BOOLEAN outside_qc_levels(qcuid; val) IS
     ~n_write_allowed(FALSE, n_qc_visibility_al(qcuid), val) OR
     ~n_write_allowed(FALSE, val, n_qc_al(qcuid));

BOOLEAN write_not_allowed(procuid; al) IS
     ~n_write_allowed(n_proc_trusted(procuid), n_proc_al(procuid), al);

BOOLEAN no_segment(procuid; voluid; seguid) IS
     IF ~n_seg_exists(voluid, seguid) THEN TRUE
     ELSE ~n_read_allowed(n_proc_trusted(procuid), n_proc_al(procuid),
                        n_seg_visibility_al(seguid));

BOOLEAN read_not_allowed(procuid; al) IS
     ~n_read_allowed(n_proc_trusted(procuid), n_proc_al(procuid), al);


     EXTERNALREFS

FROM clock:
     VFUN read_clock() -> time;
     OVFUN get_uid() -> seguid;

FROM access_levels:
     level_number max_in $(maximum level number);
     INTEGER cs_size $(category set size);
     VFUN n_read_allowed(b; al; al) -> b;
     VFUN n_write_allowed(b; al; al) -> b;

FROM processes:
     VFUN n_proc_al(procuid) -> al;
     VFUN n_proc_trusted(procuid) -> b;

FROM volumes:
     VFUN n_vol_min_al(voluid) -> al;
     VFUN n_vol_mounted(voluid) -> b;

FROM quota_cells:
     VFUN n_qc_exists(voluid; qcuid) -> b;
     VFUN n_qc_visibility_al(qcuid) -> al;
     VFUN n_qc_al(qcuid) -> al;
     OFUN change_qc_refs(qcuid; i);
     OFUN change_qc_pages_used(qcuid; i);


     FUNCTIONS
```

```
OVFUN create_seg(val; voluid; qcuid)[procuid] -> seguid;
     $(creates a new segment)
     EXCEPTIONS
          unmounted_volume(procuid, voluid);
          no_quota_cell(procuid, voluid, qcuid);
          outside_qc_levels(qcuid, val);
          write_not_allowed(procuid, val);
     EFFECTS
          seguid = EFFECTS_OF get_uid();
          'n_seg_visibility_al(seguid) = val;
          'n_seg_qc(seguid) = qcuid;
          'n_seg_exists(voluid, seguid) = TRUE;
          EFFECTS_OF change_qc_refs(qcuid, 1);

VFUN n_seg_exists(voluid; seguid) -> b;
     $(true if segment exists)
     HIDDEN;
     INITIALLY b = FALSE;

VFUN n_seg_visibility_al(seguid) -> al;
     $(returns access level of segment creator)
     HIDDEN;
     INITIALLY al = ?;

VFUN seg_visibility_al(voluid; seguid)[procuid] -> al;
     $(external form of n_seg_visibility_al)
     EXCEPTIONS
          unmounted_volume(procuid, voluid);
          no_segment(procuid, voluid, seguid);
     DERIVATION n_seg_visibility_al(seguid);

VFUN n_seg_al(seguid) -> al;
     $(returns access level of segment)
     HIDDEN;
     DERIVATION n_qc_al(n_seg_qc(seguid));

VFUN seg_al(voluid; seguid)[procuid] -> al;
     $(external form of n_seg_al)
     EXCEPTIONS
          unmounted_volume(procuid, voluid);
          no_segment(procuid, voluid, seguid);
     DERIVATION n_seg_al(seguid);

VFUN n_seg_qc(seguid) -> qcuid;
     $(returns quota cell uid of segment)
     HIDDEN;
     INITIALLY qcuid = ?;

VFUN seg_qc(voluid; seguid)[procuid] -> qcuid;
     $(external form of n_seg_qc)
     EXCEPTIONS
          unmounted_volume(procuid, voluid);
```

```
            no_segment(procuid, voluid, seguid);
            read_not_allowed(procuid, h_seg_visibility_al(seguid));
        DERIVATION h_seg_qc(seguid);

    OPUN delete_seg(voluid; seguid)[procuid];
        $(deletes a segment)
        DEFINITIONS
            access_level al IS h_seg_visibility_al(seguid);
            quota_cell_uid qcuid IS h_seg_qc(seguid);
        EXCEPTIONS
            unmounted_volume(procuid, voluid);
            no_segment(procuid, voluid, seguid);
            write_not_allowed(procuid, al);
        EFFECTS
            EFFECTS_OF truncate_seg(voluid, seguid, 0, procuid);
            h_seg_exists(voluid, seguid) = FALSE;
            EFFECTS_OF change_qc_refs(qcuid, -1);

    OPUN write_seg(seguid; offset; word);
        $(writes a word of a segment)
        DEFINITIONS
            INTEGER pageno IS (offset + page_size) / page_size;
        EFFECTS
            'h_seg_used(seguid) = TRUE;
            'h_seg_modified(seguid) = TRUE;
            'h_seg_contents(seguid, offset) = word;
            'h_page_exists(seguid, pageno) = TRUE;
            ~h_page_exists(seguid, pageno) =>
            EFFECTS_OF change_qc_pages_used(h_seg_qc(seguid), 1);

    OVPUN read_seg(procuid; seguid; offset) -> word;
        $(reads a word of a segment)
        EFFECTS
            h_write_allowed(FALSE, h_proc_al(procuid), h_seg_al(seguid))
            => 'h_seg_used(seguid) = TRUE;
            word = h_seg_contents(seguid, offset);

    VFUN h_seg_contents(seguid; offset) -> word;
        $(returns a word of a segment)
        HIDDEN;
        INITIALLY word = 0;

    VFUN h_page_exists(seguid; pageno) -> b;
        $(true if page exists)
        HIDDEN;
        INITIALLY b = FALSE;

    VFUN existing_pages(voluid; seguid)[procuid] -> bmap;
        $(returns bit map of existing pages of segment)
        EXCEPTIONS
            unmounted_volume(procuid, voluid);
            no_segment(procuid, voluid, seguid);
            read_not_allowed(procuid, h_seg_al(seguid));
```

```
     DERIVATION VECTOR (FOR i FROM 1 TO max_pageno:
                              n_page_exists(seguid, i));

OPUN truncate_seg(voluid; seguid; pageno)[procuid];
     $(truncates a segment to pageno pages)
     EXCEPTIONS
          unmounted_volume(procuid, voluid);
          no_segment(procuid, voluid, seguid);
          write_not_allowed(procuid, n_seg_al(seguid));
     EFFECTS
          FORALL i | i > pageno AND i <= max_pageno :
                    n_page_exists(seguid, i) = FALSE;
          EFFECTS OF change_qc_pages_used(n_seg_qc(seguid),
                     -total_pages(seguid, pageno + 1));
          FORALL i | i > pageno * page_size AND
                     i <= max_pageno * page_size :
                     'n_seg_contents(seguid, i) = 0;

VFUN n_seg_used(seguid) -> b;
     $(true if segment has been used since last reset)
     HIDDEN;
     INITIALLY b = FALSE;

VFUN n_seg_modified(seguid) -> b;
     $(true if segment has been modified since last reset)
     HIDDEN;
     INITIALLY b = FALSE;

VFUN n_seg_atu(seguid) -> time;
     $(returns date-time used for segment)
     HIDDEN;
     INITIALLY time = 0;

VFUN n_seg_atm(seguid) -> time;
     $(returns date-time modified for segment)
     HIDDEN;
     INITIALLY time = 0;

VFUN seg_atu_atm(voluid; seguid)[procuid] -> tt;
     $(external form of n_seg_atu and n_seg_atm)
     DEFINITIONS
          INTEGER atu IS IF n_seg_used(seguid) THEN read_clock()
               ELSE n_seg_atu(seguid);
          INTEGER atm IS IF n_seg_modified(seguid) THEN read_clock()
               ELSE n_seg_atm(seguid);
     EXCEPTIONS
          unmounted_volume(procuid, voluid);
          no_segment(procuid, voluid, seguid);
          read_not_allowed(procuid, n_seg_al(seguid));
     DERIVATION VECTOR (atu, atm);

OPUN change_atu_atm(seguid);
     $(updates atu and atm)
```

```
      EFFECTS
            n_seg_used(seguid) => 'n_seg_dtu(seguid) = read_clock();
            n_seg_modified(seguid) => 'h_seg_dtm(seguid) = read_clock();
            'n_seg_used(seguid) = FALSE;
            'n_seg_modified(seguid) = FALSE;

OPUN update_dtu_dtm(voluid; seguid)[procuid];
      $(external form of change_dtu_dtm)
      EXCEPTIONS
            unmounted_volume(procuid, voluid);
            no_segment(procuid, voluid, seguid);
            write_not_allowed(procuid, h_seg_al(seguid));
      EFFECTS
            EFFECTS_OF change_dtu_dtm(seguid);


END_MODULE
```

CHAPTER XI

ADDRESS SPACES

The address_spaces module supports the binding of segment numbers to segments within a process as well as the granting and revoking of current access to segments. This module also provides kernel interface functions for reading and writing segments.

The binding of a segment number to a segment is accomplished by the "assign_segno" function. Note that the kernel does not select the segment number. This selection is performed by the supervisor, thus relieving the kernel of a number of bookkeeping chores. Information about assigned segments is kept in a per-process table called the known segment table (KST). The KST is represented by a collection of V-functions.

The assignment of a segment number to a segment does not make the segment directly accessible. This is accomplished by another function called "give_access" for which the caller specifies the access mode, ring brackets, call limiter, and maximum length of the segment. The requested access mode is modified, if necessary, by the kernel to conform to access level restrictions. Note that any process can grant itself access to a segment in this fashion. However, it is expected that the supervisor will control the use of the give access function so as to enforce the Multics access control list (ACL) policy. Also, the supervisor will enforce the ring bracket policy. The kernel insists only that segment ring brackets be outside the kernel ring.

A function is provided to revoke the current access of all processes to a given segment. This function can be used by the supervisor to force all processes to recompute access to a segment after an ACL has been changed. Once access has been revoked, subsequent references to the segment will cause an exception. This exception is implemented as a fault that will be directed to the supervisor. In handling this fault, the supervisor will calculate the access of the faulting process to the segment and then invoke the give_access function. After this, execution can be resumed from the point at which the fault occurred.

The address_spaces module includes the revoke_vol_access function which is hidden from the kernel interface. At the time a volume is dismounted, access to all segments on the volume is revoked by use of this function. Thus, if the volume should later be remounted, all processes will be forced to recompute access to segments on the volume. This permits a user to change the ACL of a segment on a dismounted volume and

be assured that his change will take effect as soon as the volume is remounted.

Three abstract functions, called "read", "write", and "fetch", are specified in the address_spaces module. These functions do not directly correspond to any actual kernel interfaces. Rather, they represent the memory reference operations associated with whole classes of machine instructions. For example, the read function is indicative of all load instructions whereas the write function is indicative of all store instructions. The fetch function is used by the abstract machine interpreter to fetch instructions.

```
MODULE address_spaces

    TYPES

level_number : {INTEGER ln | 0 <= ln AND ln <= max_ln};
category_set : {VECTOR_OF BOOLEAN cs | LENGTH(cs) = cs_size};
access_level : STRUCT (level_number sln;
                       category_set scs;
                       level_number iln;
                       category_set ics);
process_uid : INTEGER;
volume_uid : INTEGER;
quota_cell_uid : INTEGER;
segment_uid : INTEGER;
access_mode : {VECTOR_OF BOOLEAN am | LENGTH(am) = 3};
ring_number : {INTEGER rn | 0 <= rn AND rn <= max_ring};
ring_brackets : {VECTOR_OF ring_number rb | LENGTH(rb) = 3};
call_limiter : {INTEGER cl | -1 <= cl AND cl <= max_cl};
segment_number : {INTEGER sn | 0 <= sn AND sn <= max_segno};
segment_offset : {INTEGER so | 0 <= so AND so <= max_offset};
machine_word : INTEGER;

    DECLARATIONS

process_uid procuid, proc;
volume_uid voluid;
quota_cell_uid qcuid;
segment_uid seguid;
access_level al;
access_mode mode, given_mode;
ring_number ring;
ring_brackets rb;
call_limiter cl;
segment_number segno;
segment_offset offset;
machine_word word;
BOOLEAN b;
INTEGER i;

    PARAMETERS

ring_number kernel_ring $(highest ring used by kernel);
ring_number max_ring $(maximum ring number);
call_limiter max_cl $(maximum call limiter);
segment_number max_segno $(maximum segment number);

    DEFINITIONS

access_mode null_mode IS VECTOR (FALSE, FALSE, FALSE);
```

```
access_mode rew_mode IS VECTOR (TRUE, TRUE, TRUE);
access_mode re_mode IS VECTOR (TRUE, TRUE, FALSE);

BOOLEAN write_not_allowed(procuid; al) IS
     ~h_write_allowed(h_proc_trusted(procuid), h_proc_al(procuid), al);

BOOLEAN unmounted_volume(procuid; voluid) IS
     IF ~h_vol_mounted(voluid) THEN TRUE
     ELSE ~h_read_allowed(h_proc_trusted(procuid), h_proc_al(procuid),
                          h_vol_min_al(voluid));

BOOLEAN no_segment(procuid; voluid; seguid) IS
     IF ~h_seg_exists(voluid, seguid) THEN TRUE
     ELSE ~h_read_allowed(h_proc_trusted(procuid), h_proc_al(procuid),
                          h_seg_visibility_al(seguid));

BOOLEAN segno_in_use(procuid; segno) IS
     ~(h_kst_seguid(procuid, segno) = ?);

BOOLEAN unused_segno(procuid; segno) IS
     h_kst_seguid(procuid, segno) = ?;

BOOLEAN invalid_mode(mode) IS
     ~(mode = null_mode OR mode[1]);

BOOLEAN invalid_ring_brackets(rb) IS
     ~(kernel_ring < rb[1] AND
       rb[1] <= rb[2] AND
       rb[2] <= rb[3]);

BOOLEAN no_write_permission(procuid; ring; segno) IS
     ~(h_kst_mode(procuid, segno)[3] AND
       ring <= h_kst_rb(procuid, segno)[1]);

BOOLEAN no_read_permission(procuid; ring; segno) IS
     ~(h_kst_mode(procuid, segno)[1] AND
       ring <= h_kst_rb(procuid, segno)[2]);

BOOLEAN no_execute_permission(procuid; ring; segno) IS
     ~(h_kst_mode(procuid, segno)[2] AND
       ring <= h_kst_rb(procuid, segno)[2] AND
       ring >= h_kst_rb(procuid, segno)[1]);

BOOLEAN undefined_access(procuid; segno) IS
     ~h_kst_access_defined(procuid, segno);

BOOLEAN out_of_bounds(procuid; segno; offset) IS
     offset > h_kst_max_offset(procuid, segno);

BOOLEAN page_quota_overflow(qcuid) IS
     h_qc_pages_used(qcuid) = h_qc_pages(qcuid);
```

```
     EXTERNALREFS

FROM access_levels:
     level_number max_ln $(maximum level number);
     INTEGER cs_size $(category set size);
     VFUN n_read_allowed(b; al; al) -> b;
     VFUN n_write_allowed(b; al; al) -> b;
     VFUN n_read_write_allowed(b; al; al) -> b;

FROM processes:
     VFUN n_proc_exists(procuid) -> b;
     VFUN n_proc_al(procuid) -> al;
     VFUN n_proc_trusted(procuid) -> b;

FROM volumes:
     VFUN n_vol_mounted(voluid) -> b;
     VFUN n_vol_min_al(voluid) -> al;

FROM quota_cells:
     VFUN n_qc_pages(qcuid) -> i;
     VFUN n_qc_pages_used(qcuid) -> i;

FROM segments:
     segment_offset max_offset $(maximum segment offset);
     VFUN n_seg_exists(voluid; seguid) -> b;
     VFUN n_seg_visibility_al(seguid) -> al;
     VFUN n_seg_al(seguid) -> al;
     VFUN n_seg_qc(seguid) -> qcuid;
     OFUN write_seg(seguid; offset; word);
     OVFUN read_seg(procuid; seguid; offset) -> word;
     OFUN change_stu_stm(seguid);


     FUNCTIONS

OFUN assign_segno(voluid; seguid; segno)[procuid];
     $(assigns a segment number to a segment)
     EXCEPTIONS
          segno_in_use(procuid, segno);
     EFFECTS
          'n_kst_voluid(procuid, segno) = voluid;
          'n_kst_seguid(procuid, segno) = seguid;
          'n_kst_access_defined(procuid, segno) = FALSE;

VFUN n_kst_voluid(procuid; segno) -> voluid;
     $(returns voluid of assigned segno)
     HIDDEN;
     INITIALLY voluid = ?;

VFUN kst_voluid(segno)[procuid] -> voluid;
     $(external form of n_kst_voluid)
     EXCEPTIONS
```

```
          unused_segno(segno);
     DERIVATION h_kst_voluid(procuid, segno);

VFUN h_kst_seguid(procuid; segno) -> seguid;
     $(returns seguid of assigned segno)
     HIDDEN;
     INITIALLY seguid = ?;

VFUN kst_seguid(segno)[procuid] -> seguid;
     $(external form of h_kst_seguid)
     EXCEPTIONS
          unused_segno(procuid, segno);
     DERIVATION h_kst_seguid(procuid, segno);

VFUN h_kst_access_defined(procuid; segno) -> b;
     $(true if access defined to segment for process)
     HIDDEN;
     INITIALLY b = FALSE;

OVFUN give_access(segno; offset; mode; rb; cl)[procuid] -> given_mode;
     $(gives a process access to a segment and returns given mode)
     DEFINITIONS
          volume_uid voluid IS h_kst_voluid(procuid, segno);
          segment_uid seguid IS h_kst_seguid(procuid, segno);
          access_level al IS h_seg_al(seguid);
          BOOLEAN readable IS h_read_allowed(h_proc_trusted(procuid),
                                             h_proc_al(procuid), al);
          BOOLEAN writable IS h_read_write_allowed(
                    h_proc_trusted(procuid), h_proc_al(procuid), al);
          access_mode max_mode IS IF writable THEN rew_mode
                               ELSE IF readable THEN re_mode
                               ELSE null_mode;
     EXCEPTIONS
          invalid_mode(mode);
          invalid_ring_brackets(rb);
          unused_segno(procuid, segno);
          unmounted_volume(procuid, voluid);
          no_segment(procuid, voluid, seguid);
     EFFECTS
          given_mode = VECTOR (FOR i FROM 1 TO 3:
                               mode[i] AND max_mode[i]);
          'h_kst_max_offset(procuid, segno) = offset;
          'h_kst_mode(procuid, segno) = given_mode;
          'h_kst_rb(procuid, segno) = rb;
          'h_kst_cl(procuid, segno) = cl;
          'h_kst_access_defined(procuid, segno) = TRUE;

VFUN h_kst_max_offset(procuid; segno) -> offset;
     $(maximum offset that can be referenced in segment)
     HIDDEN;
     INITIALLY offset = ?;

VFUN kst_max_offset(segno)[procuid] -> offset;
```

```
      $(external form of h kst max offset)
      EXCEPTIONS
           unused segno(procuid, segno);
           undefined access(procuid, segno);
      DERIVATION h kst max offset(procuid, segno);

VFUN h kst mode(procuid; segno) -> mode;
      $(returns access mode to segment for process)
      HIDDEN;
      INITIALLY mode = ?;

VFUN kst mode(segno)[procuid] -> mode;
      $(external form of h kst mode)
      EXCEPTIONS
           unused segno(procuid, segno);
           undefined access(procuid, segno);
      DERIVATION h kst mode(procuid, segno);

VFUN h kst rb(procuid; segno) -> rb;
      $(returns ring brackets of segment for process)
      HIDDEN;
      INITIALLY rb = ?;

VFUN kst rb(segno)[procuid] -> rb;
      $(external form of h kst rb)
      EXCEPTIONS
           unused segno(procuid, segno);
           undefined access(procuid, segno);
      DERIVATION h kst rb(procuid, segno);

VFUN h kst cl(procuid; segno) -> cl;
      $(returns call limiter of segment for process)
      HIDDEN;
      INITIALLY cl = ?;

VFUN kst cl(segno)[procuid] -> cl;
      $(external form of h kst cl)
      EXCEPTIONS
           unused segno(procuid, segno);
           undefined access(procuid, segno);
      DERIVATION h kst cl(procuid, segno);

OPUN revoke access(voluid; seguid)[procuid];
      $(revokes access to segment from all processes
         forces access to be recomputed)
      EXCEPTIONS
           unmounted volume(procuid, voluid);
           no segment(procuid, voluid, seguid);
           write not allowed(procuid, h seg al(seguid));
      EFFECTS
           FORALL proc | h proc exists(proc) :
           (FORALL segno | h kst seguid(proc, segno) = seguid :
           'h kst access defined(proc, segno) = FALSE);
```

```
OPUN revoke_vol_access(voluid; procuid);
     $(revokes access to all segments on a volume)
     EFFECTS
          FORALL seguid | h_seg_exists(voluid, seguid) :
          EFFECTS OF revoke_access(voluid, seguid, procuid);

OPUN release_segno (segno)[procuid];
     $(releases a segment number)
     DEFINITIONS
          segment_uid seguid IS h_kst_seguid(procuid, segno);
     EXCEPTIONS
          unused_segno(procuid, segno);
     EFFECTS
          'h_kst_seguid(procuid, segno) = ?;
          h_write_allowed(h_proc_trusted(procuid), h_proc_al(procuid),
                          h_seg_al(seguid)) =>
          EFFECTS OF change_stu_dtm(seguid);

OPUN purge_address_space(proc);
     $(releases all segments in address space of process)
     EFFECTS
          FORALL segno | h_kst_seguid(proc, segno) ~= ? :
          EFFECTS OF release_segno(segno, proc);

OPUN write(segno; offset; word)[procuid; ring];
     $(writes a word of a segment)
     DEFINITIONS
          volume_uid voluid IS h_kst_voluid(procuid, segno);
          segment_uid seguid IS h_kst_seguid(procuid, segno);
          quota_cell_uid qcuid IS h_seg_qc(seguid);
     EXCEPTIONS
          unused_segno(procuid, segno);
          unmounted_volume(procuid, voluid);
          no_segment(procuid, voluid, seguid);
          undefined_access(procuid, segno);
          no_write_permission(procuid, ring, segno);
          out_of_bounds(procuid, segno, offset);
          page_quota_overflow(qcuid);
     EFFECTS
          EFFECTS OF write_seg(seguid, offset, word);

OVPUN read(segno; offset)[procuid; ring] -> word;
     $(reads a word from a segment)
     DEFINITIONS
          volume_uid voluid IS h_kst_voluid(procuid, segno);
          segment_uid seguid IS h_kst_seguid(procuid, segno);
     EXCEPTIONS
          unused_segno(procuid, segno);
          unmounted_volume(procuid, voluid);
          no_segment(procuid, voluid, seguid);
          undefined_access(procuid, segno);
          no_read_permission(procuid, ring, segno);
```

```
        out_of_bounds(procuid, segno, offset);
    EFFECTS
        EFFECTS_OF read_seg(procuid, seguid, offset) = word;

OVFUN fetch(segno; offset)[procuid; ring] -> word;
    $(fetches an instruction word from a segment)
    DEFINITIONS
        volume_uid voluid IS h_kst_voluid(procuid, segno);
        segment_uid seguid IS h_kst_seguid(procuid, segno);
    EXCEPTIONS
        unused_segno(procuid, segno);
        unmounted_volume(procuid, voluid);
        no_segment(procuid, voluid, seguid);
        undefined_access(procuid, segno);
        no_execute_permission(procuid, ring, segno);
        out_of_bounds(procuid, segno, offset);
    EFFECTS
        EFFECTS_OF read_seg(procuid, seguid, offset) = word;


END_MODULE
```

CHAPTER XII

MESSAGE SEGMENTS

A message segment is a special type of segment that serves as a container for smaller objects called messages. Originally, it was envisioned that a single message segment could hold messages of various access levels. This was to be accomplished by establishing a minimum and maximum access level for each message segment. Messages within a given message segment could then be assigned distinct access levels within the allowed access range. This, in fact, is essentially the case in current Multics. Unfortunately, however, this method of sharing a message segment among multiple access levels produces another instance of the shared finite resource problem.

One approach has been suggested [Sch] that restricts message segments to a single access level, but provides an "add up" capability, i.e. allows a process to add messages to message segments of equal or greater access levels. Unfortunately, under this scheme, a process could not, in general, read messages that it had added. This would create incompatibilities in a number of message segment applications. Most noticeably, it would prohibit certain users from listing their pending qprint and absentee requests. Of course, no users would be affected at single access level sites.

It would appear that a more compatible solution (i.e. one that supports multiple access level message segments) requires a cumbersome mechanism for partitioning message segment space among different access levels. Such a mechanism cannot be justified within the kernel since the same effect can be achieved outside the kernel without substantially greater difficulty. Outside the kernel, a multiple access level message segment could be simulated using a collection of ordinary single access level segments.

As yet, no decision has been reached as to which approach to pursue. None of the alternatives seem especially pleasing. No specifications for message segments are provided at this time.

CHAPTER XIII

CONCLUSIONS

This report has presented a formal top-level specification for a Multics security kernel. The specification has been analyzed from the standpoint of security as defined by a mathematical model and also from the standpoint of compatibility with the current Multics.

With respect to security, a number of specific problems have been identified. Most of these are instances of the shared finite resource problem. Although solutions to these problems are known, they often yield awkward or incompatible interfaces and sometimes require excessive mechanism to implement. In some cases, all known alternatives have distinct disadvantages and subjective choices must be made.

Compatibility with current Multics can be achieved in most, but not all, aspects of the kernel interface. A number of incompatibilities have been introduced for reasons of security. These incompatibilities are significant, but hopefully tolerable in an environment where security is the foremost objective. Also, another category of incompatibilities have been introduced to promote simplicity and thereby facilitate the proof of security properties. These incompatibilities can be eliminated at a cost of increased verification difficulty.

The formal specifications presented here represent only an initial effort. They are not yet complete nor fully refined. However, this first step has been generally encouraging. The specification technique has proved helpful in illuminating a variety of issues. The remaining problems now seem, at least, specific and well understood. Even without further enhancement, the top-level specification presented here should serve as a useful guide for related work in proving security properties and developing lower level kernel specifications.

REFERENCES

[And] J.P. Anderson, Computer Security Technology Planning Study, ESD-TR-73-51, Vol II, Electronic Systems Division (AFSC), L.G. Hanscom Field, Bedford, Mass., October 1972

[BL] D.E. Bell and L.J. LaPadula, Computer Security Model: Unified Exposition and Multics Interpretation, MTR-2997, The MITRE Corporation, Bedford, Mass., June 1975

[Ben] A. Bensoussan, "Removing Directory Control from the Multics Security Kernel", Project Guardian TCL-17, Honeywell, March 1976

[Bib] K.J. Biba, Integrity Considerations for Secure Computer Systems, MTR-3153, The MITRE Corporation, Bedford, Mass., June 1975

[BR1] R.S. Boyer and L. Robinson, "Abstract Machine Interpreters and Parallelism", Project Guardian TCL to appear, Stanford Research Institute, January 1976

[BR2] R.S. Boyer and L. Robinson, "Parallelism and the Semantics of Indivisibility", Project Guardian TCL to appear, Stanford Research Institute, April 1976

[Hon] External I/O in a Secure Kernel-Based Multics Computer System, to be published, Honeywell, Cambridge, Mass.

[Jan] P. Janson, private communication

[Mil] J.K. Millen, "Security Kernel Validation in Practice", CACM, Volume 19, Number 5, May 1976, pp. 243-250

[Par] D.L. Parnas, "A Technique for Software Module Specification with Examples", CACM, Volume 15, Number 5, May 1972, pp. 330-336

[Rob] L. Robinson, "The Importance of Abstract Machine Interpreters", Project Guardian TCL to appear, Stanford Research Institute, January 1976

[RLNS] L. Robinson, K.N. Levitt, P.G. Neumann, and A.R. Saxena, "On Attaining Reliable Software for a Secure Operating System", 1975 Conference on Reliable Software, Los Angeles, California, April, 1975, pp. 267-284

APPENDIX A

Air Force Electronic Systems Division Comments

General - The report is not complete. Many areas of the design are omitted such as message segments, external I/O, reconfiguration, utilization, and "trusted subject" interfaces. Specific design decisions are avoided in certain areas, most notably various instances of the "finite resource problem". Most of the omissions are explicitly acknowledged in the report.

Page 1, line -2 (2nd line from bottom of page). The reference monitor is not restricted to enforcing rules of the military security system.

Page 2, line +10 (10th line from top of page). In the ESD security literature, "verified" and "certified" have had distinct meanings and are not interchangeable. Verification is a technical activity; certification is an administrative action available only to a designated authority. Verification provides partial justification for certification.

Page 2, line -5. Some mention of the relative priority of access control vs. usability/compatibility is needed.

Page 3, line +10. The physical size of the kernel is not the real issue, rather, it is the number and complexity of specified functions.

Page 3, line -22. The discussion does not explicitly address the isolation of kernel processes.

Page 3, line -15. The "special interfaces" either need not be specified at the top level or should be explicitly identified as special.

Page 4, line +6. Message segments have been omitted also.

Page 5, line +14. The model does not state that a subject is a surrogate for a user, which is not always the case.

Page 5, line -15. The notion of a visibility access level is not explicitly included in the current mathematical model. The need for a revised model should be addressed. It does not appear that the access level of the creator of an object could differ from the visibility access level of the object.

Page 6, line +13. The nature of the kernel maintained "simpler representation of the access matrix for segment objects" should be described in more detail.

[RR] O. Roubine and L. Robinson, SPECIAL Reference Manual, Technical Report CSG-45, Stanford Research Institute, Menlo Park, California, August 1976

[Sch] W.L. Schiller, private communication

[SBB] W.L. Schiller, K.J. Biba, and E.L. Burke, The Top Level Specification of a Multics Security Kernel, WP-20377, The MITRE Corporation, Bedford, Mass., August 1975

[SS] M.D. Schroeder and J.H. Saltzer, "A Hardware Architecture for Implementing Protection Rings", CACM, Volume 15, Number 3, March 1972, pp. 157-170

[Smi] L. Smith, Architectures for Secure Computing Systems, ESD-TR-75-51, The MITRE Corporation, Bedford, Mass., June 1974

[WBCHKS] J. Whitmore, A. Bensoussan, P. Green, D. Hunt, A. Kobziar, and J. Stern, Design for Multics Security Enhancements, ESD-TR-74-176, L.G. Hanscom Field, Bedford, Mass., October 1973

APPENDIX A

Air Force Electronic Systems Division Comments

General - The report is not complete. Many areas of the design are omitted such as message segments, external I/O, reconfiguration, utilization, and "trusted subject" interfaces. Specific design decisions are avoided in certain areas, most notably various instances of the "finite resource problem". Most of the omissions are explicitly acknowledged in the report.

Page 1, line -2 (2nd line from bottom of page). The reference monitor is not restricted to enforcing rules of the military security system.

Page 2, line +10 (10th line from top of page). In the ESD security literature, "verified" and "certified" have had distinct meanings and are not interchangeable. Verification is a technical activity; certification is an administrative action available only to a designated authority. Verification provides partial justification for certification.

Page 2, line -5. Some mention of the relative priority of access control vs. usability/compatibility is needed.

Page 3, line +10. The physical size of the kernel is not the real issue, rather, it is the number and complexity of specified functions.

Page 3, line -22. The discussion does not explicitly address the isolation of kernel processes.

Page 3, line -15. The "special interfaces" either need not be specified at the top level or should be explicitly identified as special.

Page 4, line +6. Message segments have been omitted also.

Page 5, line +14. The model does not state that a subject is a surrogate for a user, which is not always the case.

Page 5, line -15. The notion of a visibility access level is not explicitly included in the current mathematical model. The need for a revised model should be addressed. It does not appear that the access level of the creator of an object could differ from the visibility access level of the object.

Page 6, line +13. The nature of the kernel maintained "simpler representation of the access matrix for segment objects" should be described in more detail.

Page 8, line -18. The third approach for handling finite resources differs from the other two in that it is a mechanism rather than policy. The trusted process could enforce either the quota per level or the time-multiplexing method. An alternative third policy could be to provide audit and surveillance tools to report requests for resources in excess of that available so that judgment can be made and action taken if it appears that users are circumventing security controls through finite resource limitations. Actions that could be undertaken externally include adding new resources to the system, freeing used resources or denying further system access to a requestor suspected of attempting to circumvent the security controls.

Page 12, line 13. The omission of exeptions for internal functions could effect verification. Certain constraints must be satisfied in order for the function to produce its effects. The exception conditions of referencing functions must insure the constraints are satisfied.

Page 15, line +17. The rationale for hiding some U and OV functions is not apparent.

Page 16, advance_clock. The use of this function is not apparent. It seems that this function need (would) not be available at the kernel interface. More natural alternatives may be: (1) include changing the value of the clock in a specification of the instruction fetch cycle (Note that the increment would be arbitrary for may "kernel instructions"). (2) have get_uid update the clock (implies the value of read_clock is only changed by calls to get_uid) and (3) change the specification of read_clock to show that the clock and kernel processes operate asynchronously (seems to be the most realistic). For example, the specification could merely state that time > 'time (or time > 'time if the granularity of the clock was finer than the time required to do the read operation).

Page 24, delete proc. Please provide the rationale for not allowing a process to delete itself.

Page 25, wake. It is not clear why this function exists since only wakeup uses it.

Page 28, line -4. This discussion does not consider the time-multiplexing approach to solving the problem. If every user had to mount (a virtual mount) a volume before the volume was referenced, there would be no storage channels.

Page 30, create volume. The initial quota for every volume is always a constant. This does not seem appropriate because the volume may consist of a variable number of physical secondary

storage devices and arbitrary areas of some devices may be unusable.

Page 33, line -17. The restriction against moving quota downward will cause quota to be lost when upgraded, terminal segments are deleted. Deleting terminal segments should get quota back to its original level. This is a deficiency in the "delete_seg" function. Quotat on an upgraded segment is lost when the segment is deleted.

Page 35, line -22. As noted, a method for controlling the number of quota cells is needed.

Page 24, line -20. A third incompatibility was noted in the comments on the previous page.

Page 41, line 12. As noted, here is another unsolved finite resource problem.

Page 41, line -5. Zero internal pages could be handled by a "free" function as well as truncate.

Page 43, line +16. Incorporation of dtu and dtm in the kernel should be justified.

Page 45, PARAMETERS. The specification implies the max_offset, max_pageno and page_size are all independent. It seems that one should be the function of the other two.

Page 47. The function parameters seem inconsistent. Some functions have both voluid and seguid as parameters while others have only seguid. Please explain.

Page 51, line -6. The need for revoke_vol_access is not clear. The revoke_seg_access will handle the example given.

Page 56, give_access. Since offset is a parameter, two processes may share parts of a segment. It seems this may complicate the management of page tables. The justification for providing this capability would be helpful.

Page 60, line -3. As noted message segments are missing.

MISSION
OF THE
DIRECTORATE OF COMPUTER SYSTEMS ENGINEERING

The Directorate of Computer Systems Engineering provides ESD with technical services on matters involving computer technology to help ESD system development and acquisition offices exploit computer technology through engineering application to enhance Air Force systems and to develop guidance to minimize R&D and investment costs in the application of computer technology.

The Directorate of Computer Systems Engineering also supports AFSC to insure the transfer of computer technology and information throughout the Command, including maintaining an overview of all matters pertaining to the development, acquisition, and use of computer resources in systems in all Divisions, Centers and Laboratories and providing AFSC with a corporate memory for all problems/solutions and developing recommendations for RDT&E programs and changes in management policies to insure such problems do not reoccur.